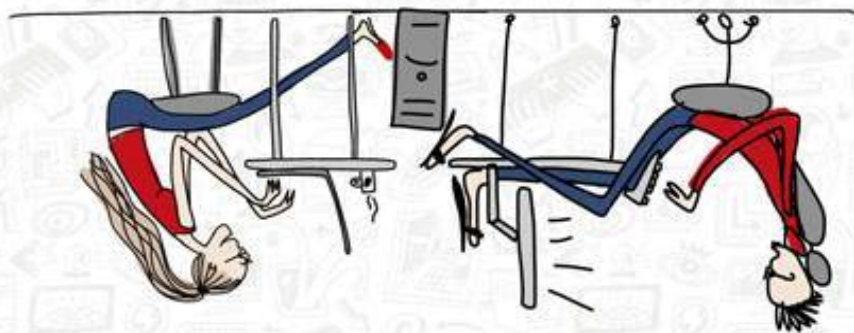




自己动手 构造编译系统

编译、汇编与链接

范志东 张琼声 著



机械工业出版社
China Machine Press

自己动手系列

自己动手构造编译系统：编译、汇编与链接

范志东 张琼声 著

ISBN: 978-7-111-54355-8

本书纸版由机械工业出版社于2016年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

序

前言

第1章 代码背后

1.1 从编程聊起

1.2 历史渊源

1.3 GCC的工作流程

1.3.1 预编译

1.3.2 编译

1.3.3 汇编

1.3.4 链接

1.4 设计自己的编译系统

1.5 本章小结

第2章 编译系统设计

2.1 编译程序的设计

2.1.1 词法分析

2.1.2 语法分析

2.1.3 符号表管理

2.1.4 语义分析

2.1.5 代码生成

2.1.6 编译优化

2.2 x86指令格式

2.3 ELF文件格式

2.4 汇编程序的设计

2.4.1 汇编词法、语法分析

2.4.2 表信息生成

2.4.3 指令生成

2.5 链接程序的设计

2.5.1 地址空间分配

2.5.2 符号解析

2.5.3 重定位

2.6 本章小结

第3章 编译器构造

3.1 词法分析

3.1.1 扫描器

3.1.2 词法记号

3.1.3 有限自动机

3.1.4 解析器

3.1.5 错误处理

3.2 语法分析

3.2.1 文法定义

- 3.2.2 递归下降子程序
 - 3.2.3 错误处理
- 3.3 符号表管理
 - 3.3.1 符号表数据结构
 - 3.3.2 作用域管理
 - 3.3.3 变量管理
 - 3.3.4 函数管理
- 3.4 语义分析
 - 3.4.1 声明与定义语义检查
 - 3.4.2 表达式语义检查
 - 3.4.3 语句语义检查
 - 3.4.4 错误处理
- 3.5 代码生成
 - 3.5.1 中间代码设计
 - 3.5.2 程序运行时存储
 - 3.5.3 函数定义与return语句翻译
 - 3.5.4 表达式翻译
 - 3.5.5 复合语句与break、continue语句翻译
 - 3.5.6 目标代码生成
 - 3.5.7 数据段生成
- 3.6 本章小结

第4章 编译优化

4.1 数据流分析

4.1.1 流图

4.1.2 数据流分析框架

4.2 中间代码优化

4.2.1 常量传播

4.2.2 复写传播

4.2.3 死代码消除

4.3 寄存器分配

4.3.1 图着色算法

4.3.2 变量栈帧偏移计算

4.4 窥孔优化

4.5 本章小结

第5章 二进制表示

5.1 x86指令

5.1.1 指令前缀

5.1.2 操作码

5.1.3 ModR/M字段

5.1.4 SIB字段

5.1.5 偏移

5.1.6 立即数

5.1.7 AT&T汇编格式

5.2 ELF文件

5.2.1 文件头

5.2.2 段表

5.2.3 程序头表

5.2.4 符号表

5.2.5 重定位表

5.2.6 串表

5.3 本章小结

第6章 汇编器构造

6.1 词法分析

6.1.1 词法记号

6.1.2 有限自动机

6.2 语法分析

6.2.1 汇编语言程序

6.2.2 数据定义

6.2.3 指令

6.3 符号表管理

6.3.1 数据结构

6.3.2 符号管理

6.4 表信息生成

6.4.1 段表信息

6.4.2 符号表信息

6.4.3 重定位表信息

6.5 指令生成

6.5.1 双操作数指令

6.5.2 单操作数指令

6.5.3 零操作数指令

6.6 目标文件生成

6.7 本章小结

第7章 链接器构造

7.1 信息收集

7.1.1 目标文件信息

7.1.2 段数据信息

7.1.3 符号引用信息

7.2 地址空间分配

7.3 符号解析

7.3.1 符号引用验证

7.3.2 符号地址解析

7.4 重定位

7.5 程序入口点与运行时库

7.6 可执行文件生成

7.7 本章小结

参考文献

序

小范从本科毕业设计开始写编译器的实现代码，为他选择这个题目的初衷是希望把编译系统与操作系统、计算机体系结构相关的结合点找出来、弄清楚，为教学提供可用的实例。本科毕业设计结束时小范完成了一个最简单的C语言子集的编译器，生成的汇编程序经过汇编和链接后可以正确执行。研究生期间我们决定继续编译系统实现技术方向的研究工作，主要完成汇编器和链接器这两大模块。小范用一颗好奇、求知的心指引自己，利用一切可以搜集到的资料，用“日拱一卒”的劲头一步一步接近目标。每天的日子都可能有不同的“干扰”——名企的实习、发论文、做项目、参加竞赛、考认证，身边的同学在快速积攒各种经历和成果的时候，小范要保持内心的平静，专注于工作量巨大而是否有回报还未曾可知的事情。三年的时间里，没有奖学金，没有项目经费，有的是没完没了的各种问题，各种要看的书、资料 and 要完成的代码，同时还要关注大数据平台、编程语言等新技术的发展。

“汇编器完成了”“链接器完成了”，好消息接踵而至。小范说，“把编译器的代码重写一下，加上代码优化吧？”我说“好”，其实，这个“好”说起来容易，而小范那里增加的工作量可想而知，这绝不是那么轻松的事情。优化的基本原理有了，怎么设计算法来实现呢？整个

编译器的文法比本科毕业设计时扩充了很多。编译器重写、增加代码优化模块、完成汇编器和链接器，难度和工作量可想而知。每当小范解决一个问题，完成一个功能，就会非常开心地与我分享。看小范完成的一行行规范、漂亮的代码，听他兴奋地讲解，很难说与听郎朗的钢琴协奏曲《黄河之子》、德沃夏克的《自新大陆》比哪一个更令人陶醉，与听交响曲《嘎达梅林》比哪一个更令人震撼。当小范完成链接器后，我说：“小范，写书吧，不写下来太可惜了。”就这样，小范再次如一辆崭新的装甲车，轰隆前行，踏上了笔耕不辍的征程。2015年暑假，细读和修改这部30多万字的书稿，感慨万千，完成编译系统的工作量、四年的甘苦与共、超然物外的孤独都在这字里行间跳跃。写完这部原创书对一个年轻学生来说是极富挑战的，但是他完成了，而且完成得如此精致、用心。

小范来自安徽的农村，面对生活中的各种困惑、困难，他很少有沮丧、悲观的情绪，永远有天然的好奇心，保留着顽童的天真、快乐与坦率。他开始写本书时23岁，完成全书的初稿时25岁。写编译系统和操作系统内核并非难以企及，只是需要一份淡然、专注和坚持。

如果你想了解计算机是如何工作的，为什么程序会出现不可思议的错误？高级语言程序是如何被翻译成机器语言代码的？编译器在程序的优化方面能做哪些工作？软件和硬件是怎么结合工作的？各种复杂的数据结构和算法，包括图论在实现编译系统时如何应用？有限自

动机在词法分析中的作用是什么？其程序又如何实现？那么本书可以满足你的好奇心和求知欲。如何实现编译系统？如何实现编译器？如何实现汇编器？如何使用符号表？如何结合操作系统加载器的需要实现链接器？Intel的指令是如何构成的？如何实现不同的编译优化算法？对这些问题，本书结合作者实现的代码实例进行了详尽的阐述，对提高程序员的专业素质有实际的助益，同时本书也可以作为计算机科学相关专业教师的参考书和编译原理实习类课程的教材。

2013年在新疆参加全国操作系统和组成原理教学研讨会时，我带着打印出来的两章书稿给了机械工业出版社的温莉芳老师，与她探讨这本书出版的意义和可行性，她给了我们很大的鼓励和支持，促成了本书的完成。在此，特别感谢温莉芳老师。

本书的责任编辑余洁老师与作者反复沟通，对本书进行了认真、耐心的编辑，感谢她的辛勤付出。

中国石油大学（华东）的李村合老师在编译器设计的初期给予了我们指导和建议。马力老师在繁忙的工作之余，认真审阅书稿，给出了详细的修改意见。王小云、程坚、梁红卫、葛永文老师对本书提出了他们的意见，并给出了认真的评价。赵国梁同学对书中的代码和文字做了细心的校对。在此，对他们表示衷心的感谢。最后要感谢小范勤劳、坚韧的爸爸妈妈，是他们一直给予他无私的支持和持续的鼓励。

感恩所有给予我们帮助和鼓励的老师、同学和朋友！

张琼声

2016年春于北京

前言

本书适合谁读

本书是一本描述编译系统实现的书籍。这里使用“编译系统”一词，主要是为了与市面上描述编译器实现的书籍进行区分。本书描述的编译系统不仅包含编译器的实现，还包括汇编器、链接器的实现，以及机器指令与可执行文件格式的知识。因此，本书使用“编译系统”一词作为编译器、汇编器和链接器的统称。

本书的目的是希望读者能通过阅读本书清晰地认识编译系统的工作流程，并能自己尝试构造一个完整的编译系统。为了使读者更容易理解和学习编译系统的构造方法，本书将描述的重点放在编译系统的关键流程上，并对工业化编译系统的实现做了适当的简化。如果读者对编译系统实现的内幕感兴趣，或者想自己动手实现一个编译系统的话，本书将非常适合你阅读。

阅读本书，你会发现书中的内容与传统的编译原理教材以及描述编译器实现的书籍有所不同。本书除了描述一个编译器的具体实现外，还描述了一般书籍较少涉及的汇编器和链接器的具体实现。而且本书并非“纸上谈兵”，在讲述每个功能模块时，书中都会结合具体实

现代码来阐述模块功能的实现。通过本书读者将会学习如何使用有限自动机构造词法分析器，如何将文法分析算法应用到语法分析过程，如何使用数据流分析进行中间代码的优化，如何生成合法的汇编代码，如何产生二进制指令信息，如何在链接器内进行符号解析和重定位，如何生成目标文件和可执行文件等。

本书的宗旨是为意欲了解或亲自实现编译系统的读者提供指导和帮助。尤其是计算机专业的读者，通过自己动手写出一个编译系统，能加强读者对计算机系统从软件层次到硬件层次的理解。同时，深入挖掘技术幕后的秘密也是对专业兴趣的一种良好培养。GCC本身是一套非常完善的工业化编译系统（虽然我们习惯上称它为编译器），然而单凭个人之力无法做到像GCC这样完善，而且很多时候是没有必要做出一个工程化的编译器的。本书试图帮助读者深入理解编译的过程，并能按照书中的指导实现一个能正常工作的编译器。在自己亲自动手实现一个编译系统的过程中，读者获得的不仅仅是软件开发的经历。在开发编译系统的过程中，读者还会学习很多与底层相关的知识，而这些知识在一般的专业教材中很少涉及。

如果读者想了解计算机程序底层工作的奥秘，本书能够解答你内心的疑惑。如果读者想自定义一种高级语言，并希望使该语言的程序在计算机上正常运行，本书能帮助你较快地达到目的。如果读者想从

实现一个编译器的过程中，加强对编译系统工作流程的理解，并尝试深入研究GCC源码，本书也能为你提供很多有价值的参考。

基础知识储备

本书尽可能地不要求读者有太多的基础知识准备，但是编译理论属于计算机学科比较深层次的知识领域，难免对读者的知识储备有所要求。本书的编译系统是基于Linux x86平台实现的，因此要求读者对Linux环境的C/C++编程有所了解。另外，理解汇编器的实现内容需要读者对x86的汇编指令编程比较熟悉。本书不会描述过多编译原理教材中涉及的内容，所以要求读者具备编译原理的基础知识。不过读者不必过于担心，本书会按照循序渐进的方式描述编译系统的实现，在具体的章节中会将编译系统实现的每个细节以及所需的知识阐述清楚。

本书内容组织

本书共7章，各章的主要内容分别如下。

第1章 代码背后

从程序设计开始，追溯代码背后的细节，引出编译系统的概念。

第2章 编译系统设计

按照编译系统的工作流程，介绍本书编译系统的设计结构。

第3章 编译器构造

描述如何使用有限自动机识别自定义高级语言的词法记号，如何使用文法分析算法识别程序的语法模块，如何对高级语言上下文相关信息进行语义合法性检查，如何使用语法制导翻译进行代码生成，以及编译器工作时符号信息的管理等。

第4章 编译优化

介绍中间代码的设计和生成，如何利用数据流分析实现中间代码优化，如何对变量进行寄存器分配，目标代码生成阶段如何使用窥孔优化器对目标代码进行优化。

第5章 二进制表示

描述Intel x86指令的基本格式，并将AT&T汇编与Intel汇编进行对比。描述ELF文件的基本格式，介绍ELF文件的组织和操作方法。

第6章 汇编器构造

描述汇编器词法分析和语法分析的实现，介绍汇编器如何提取目标文件的主要表信息，并描述x86二进制指令的输出方法。

第7章 链接器构造

介绍如何为可重定位目标文件的段进行地址空间分配，描述链接器符号解析的流程，以及符号地址的计算方法，并介绍重定位在链接器中的实现。

随书源码

本书实现的编译系统代码已经托管到github，源码可以使用GCC 5.2.0编译通过。代码的github地址是 <https://github.com/fanzhidongyzby/cit>。代码分支x86实现了基于Intel x86体系结构的编译器、汇编器和链接器，编译系统生成的目标文件和可执行文件都是Linux下标准的ELF文件格式。代码分支arm实现了基于ARM体系结构的编译器，目前支持生成ARM 7的汇编代码。

第1章 代码背后

知其然，并知其所以然。

——《朱子语类》

1.1 从编程聊起

说起编程，如果有人问我们敲进计算机的第一段代码是什么，相信很多人会说出同一个答案——“Hello World！”。编程语言的教材一般都会把这段代码作为书中的第一个例子呈现给读者。当我们按照课本或者老师的要求把它输入到开发环境，然后单击“编译”和“运行”按钮，映入眼帘的那行字符串定会令人欣喜不已！然而激动过后，一股强烈的好奇心可能会驱使我们去弄清一个新的概念——编译是什么？

遗憾的是，一般教授编程语言的老师不会介绍太多关于它的内容，最多会告诉我们：代码只有经过编译，才能在计算机中正确执行。随着知识和经验的不断积累，我们逐渐了解到当初单击“编译”按钮的时候，计算机在幕后做了一系列的工作。它先对源代码进行编译，生成二进制目标文件，然后对目标文件进行链接，最后生成一个可执行文件。即便如此，我们对编译的流程也只有一个模糊的认识。

直到学习了编译原理，才发现编译器原来就是语言翻译程序，它把高级语言程序翻译成低级汇编语言程序。而汇编语言程序是不能被计算机直接识别的，必须靠汇编器把它翻译为计算机硬件可识别的机器语言程序。而根据之前对目标文件和链接器的了解，我们可能猜测到机器语言应该是按照二进制的形式存储在目标文件内部的。可是目

标文件到底包含什么，链接后的可执行文件里又有什么？问题貌似越来越多。

图1-1展示了编译的大致工作流程，相信拥有一定编程经验的人，对该图所表达的含义并不陌生。为了让源代码能正常地运行在计算机上，计算机对代码进行了“繁复”的处理。可是，编译器既然是语言翻译程序，为什么不把源代码直接翻译成机器语言，却还要经过汇编和链接的过程呢？

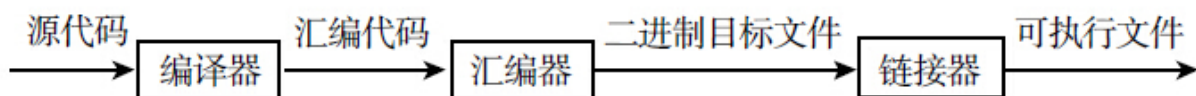


图1-1 编译的流程

似乎我们解决了一些疑惑后，总是会有更多的疑惑接踵而来。但也正是这些层出不穷的疑惑，促使我们不断地探究简单问题背后的复杂机制。当挖掘出这些表象下覆盖的问题本质时，可能比首次敲出“Hello World！”程序时还要喜悦。在后面的章节中，将会逐步探讨编译背后的本质，将谜团一一揭开，最终读者自己可动手构造出本书所实现的编译系统——编译器、汇编器与链接器，真正做到“知其然，并知其所以然”。

1.2 历史渊源

历史上很多新鲜事物的出现都不是偶然的，计算机学科的技术和知识如此，编译系统也不例外，它的产生来源于编程工作的需求。编程本质上是人与计算机交流，人们使用计算机解决问题，必须把问题转化为计算机所能理解的方式。当问题规模逐渐增大时，编程的劳动量自然会变得繁重。编译系统的出现在一定程度上降低了编程的难度和复杂度。

在计算机刚刚诞生的年代，人们只能通过二进制机器指令指挥计算机工作，计算机程序是依靠人工拨动计算机控制面板上的开关被输入到计算机内部的。后来人们想到使用穿孔卡片来代替原始的开关输入，用卡片上穿孔的有无表示计算机世界的“0”和“1”，让计算机自动读取穿孔卡片实现程序的录入，这里录入的指令就是常说的二进制代码。然而这种编程工作在现在看起来简直就是一个“噩梦”，因为一旦穿孔卡片的制作出现错误，所有的工作都要重新来过。

人们很快就发现了使用二进制代码控制计算机的不足，因为人工输入二进制指令的错误率实在太高了。为了解决这个问题，人们用一系列简单明了的助记符代替计算机的二进制指令，即我们熟知的汇编语言。可是计算机只能识别二进制指令，因此需要一个已有的程序自

动完成汇编语言到二进制指令的翻译工作，于是汇编器就产生了。程序员只需要写出汇编代码，然后交给汇编器进行翻译，生成二进制代码。因此，汇编器将程序员从烦琐的二进制代码中解脱出来。

使用汇编器提高了编程的效率，使得人们有能力处理更复杂的计算问题。随着计算问题复杂度的提高，编程中出现了大量的重复代码。人们不愿意进行重复的劳动，于是就想办法将公共的代码提取出来，汇编成独立的模块存储在目标文件中，甚至将同一类的目标文件打包成库。由于原本写在同一个文件内的代码被分割到多个文件中，那么最终还需要将这些分离的文件拼装起来形成完整的可执行代码。但是事情并没有那么简单，由于文件的模块化分割，文件间的符号可能会相互引用。人们需要处理这些引用关系，重新计算符号的引用地址，这就是链接器的基本功能。链接器使得计算机能自动把不同的文件模块准确无误地拼接起来，使得代码的复用成为可能。

图1-2描述的链接方式称为静态链接，但这种方式也有不足之处。静态链接器把公用库内的目标文件合并到可执行文件内部，使得可执行文件的体积变得庞大。这样做会导致可执行文件版本难以更新，也导致了多个程序加载后相同的公用库代码占用了多份内存空间。为了解决上述的问题，现代编译系统都引入了动态链接方式（见图1-3）。动态链接器不会把公用库内的目标文件合并到可执行文件内，而仅仅记录动态链接库的路径信息。它允许程序运行前才加载所需的动态链

接库，如果该动态链接库已加载到内存，则不需要重复加载。另外，动态链接器也允许将动态链接库的加载延迟到程序执行库函数调用的那一刻。这样做，不仅节约了磁盘和内存空间，还方便了可执行文件版本的更新。如果应用程序模块设计合理的话，程序更新时只需要更新模块对应的动态链接库即可。当然，动态链接的方式也有缺点。运行时链接的方式会增加程序执行的时间开销。另外，动态链接库的版本错误可能会导致程序无法执行。由于静态链接和动态链接的基本原理类似，且动态链接器的实现相对复杂，因此本书编译系统所实现的链接器采用静态链接的方式。

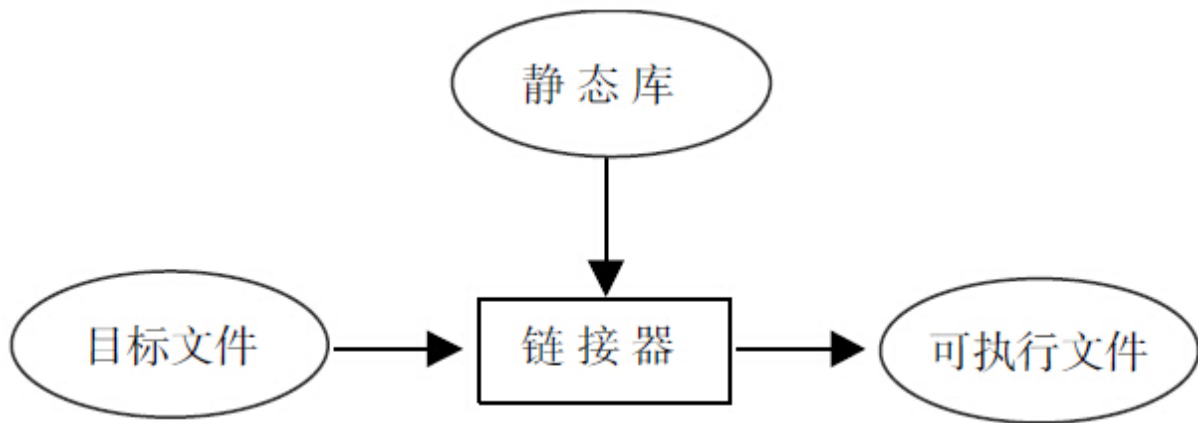


图1-2 静态链接

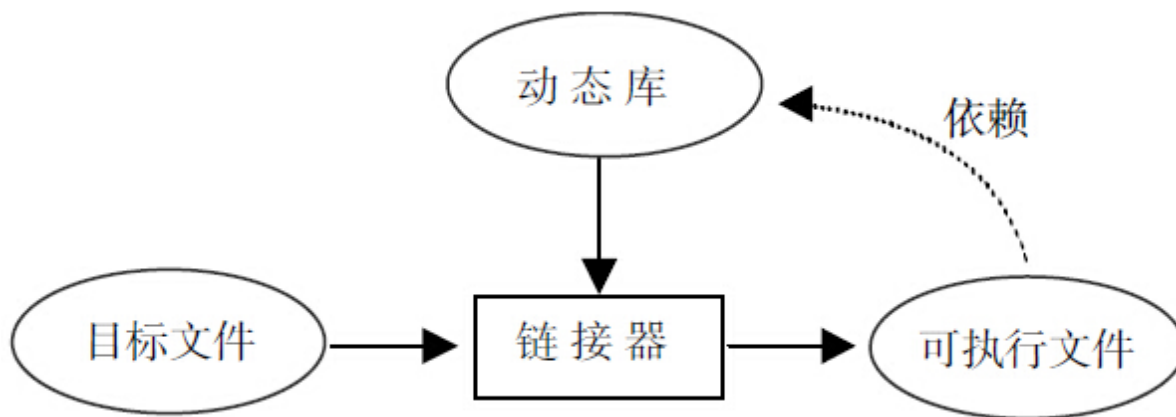


图1-3 动态链接

汇编器和链接器的出现大大提高了编程效率，降低了编程和维护的难度。但是人们对汇编语言的能力并不满足，有人设想要是能像写数学公式那样对计算机编程就太方便了，于是就出现了如今形形色色的高级编程语言。这样就面临与当初汇编器产生时同样的问题——如何将高级语言翻译为汇编语言，这正是编译器所做的工作。编译器比汇编器复杂得多。汇编语言的语法比较单一，它与机器语言有基本的对应关系。而高级语言形式比较自由，计算机识别高级语言的含义比较困难，而且它的语句翻译为汇编语言序列时有多种选择，如何选择更好的序列作为翻译结果也是比较困难的，不过最终这些问题都得以解决。高级语言编译器的出现，实现了人们使用简洁易懂的编程语言与计算机交流的目的。

1.3 GCC的工作流程

在着手构造编译系统之前，需要先介绍编译系统应该做的事情，而最具参考价值的资料就是主流编译器的实现。GNU的GCC编译器是工业化编译器的代表，因此我们先了解GCC都在做什么。

我们写一个最简单的“HelloWorld”程序，代码存储在源文件hello.c中，源文件内容如下：

```
#include<stdio.h>
int main()
{
    printf("Hello World!");
    return 0;
}
```

如果将hello.c编译并静态链接为可执行文件，使用如下gcc命令直接编译即可：

```
$gcc hello.c -
o hello -static
```

hello即编译后的可执行文件。

如果查看GCC背后的工作流程，可以使用--verbose选项。

```
$gcc hello.c -
o hello -
```

```
static --verbose
```

输出的信息如下:

```
$cc1 -quiet hello.c -o hello.s
$as -o hello.o hello.s
$collect2 -static -o hello \
    crt1.o crti.o crtbeginT.o hello.o \
    --start-group libgcc.a libgcc_eh.a libc.a --end-group \
    crtend.o crtn.o
```

为了保持输出信息的简洁，这里对输出信息进行了整理。可以看出，GCC编译背后使用了cc1、as、collect2三个命令。其中cc1是GCC的编译器，它将源文件hello.c编译为hello.s。as是汇编器命令，它将hello.s汇编为hello.o目标文件。collect2是链接器命令，它是对命令ld的封装。静态链接时，GCC将C语言运行时库（CRT）内的5个重要的目标文件crt1.o、crti.o、crtbeginT.o、crtend.o、crtn.o以及3个静态库libgcc.a、libgcc_eh.a、libc.a链接到可执行文件hello。此外，cc1在对源文件编译之前，还有预编译的过程。

因此，我们从预编译、编译、汇编和链接四个阶段查看GCC的工作细节。

1.3.1 预编译

GCC对源文件的第一阶段的处理是预编译，主要是处理宏定义和文件包含等信息。命令格式如下：

```
$gcc -  
E hello.c -  
o hello.i
```

预编译器将hello.c处理后输出到文件hello.i，hello.i文件内容如下：

```
# 1 "hello.c"  
# 1 "<built-in>"  
# 1 "<command-line>"  
# 1 "hello.c".....  
  
extern int printf (const char *__restrict __format, ...);.....  
  
int main()  
{  
    printf("Hello World!");  
    return 0;  
}
```

比如文件包含语句#include<stdio.h>，预编译器会将stdio.h的文件内容拷贝到#include语句声明的位置。如果源文件内使用#define语句定义了宏，预编译器则将该宏的内容替换到其被引用的位置。如果宏定义本身使用了其他宏，则预编译器需要将宏递归地展开。

我们可以将预编译的工作简单地理解为源码的文本替换，即将宏定义的内容替换到宏的引用位置。当然，这样理解有一定的片面性，因为要考虑宏定义中使用其他宏的情况。事实上预编译器的实现机制和编译器有着很大的相似性，因此本书描述的编译系统将重点放在源代码的编译上，不再独立实现预编译器。然而，我们需要清楚的事实是：一个完善的编译器是需要预编译器的。

1.3.2 编译

接下来GCC对hello.i进行编译，命令如下：

```
$gcc -  
S hello.i -  
o hello.s
```

编译后产生的汇编文件hello.s内容如下:

```

.file "hello.c"
.section .rodata
.LC0:
.string
    "Hello World!"

.text
.globl main
.type main, @functionmain:

pushl %ebp
movl %esp, %ebp
andl $-16, %esp
subl $16, %esp
movl $.LC0, %eax
movl %eax, (%esp)
call printf

movl $0, %eax
leave
ret
.size main, .-main
.ident "GCC: (Ubuntu/Linaro 4.4.4-14ubuntu5) 4.4.5"
.section .note.GNU-stack,"",@progbits

```


GCC生成的汇编代码的语法是AT&T格式，与Intel格式的汇编有所不同（若要生成Intel格式的汇编代码，使用编译选项“`-masm=intel`”即可）。比如立即数用“\$”前缀，寄存器用“%”前缀，内存寻址使用小括号等。区别最大的是，AT&T汇编指令的源操作数在前，目标操作数在后，这与Intel汇编语法正好相反。本书会在后续章节中详细描述这两种汇编语法格式的区别。

不过我们仍能从中发现高级语言代码中传递过来的信息，比如字符串“Hello World!”、主函数名称main、函数调用call printf等。

1.3.3 汇编

接着，GCC使用汇编器对hello.s进行汇编，命令如下：

```
$gcc -  
c hello.s -  
o hello.o
```

生成的目标文件hello.o，Linux下称之为可重定位目标文件。目标文件无法使用文本编辑器直接查看，但是我们可以使用GCC自带的工具objdump命令分析它的内容，命令格式如下：

```
$objdump -  
sd hello.o
```

输出目标文件的主要段的内容与反汇编代码如下：

```
hello.o:      file format elf32-i386  
Contents of section .text:  
0000 5589e583 e4f083ec 10b80000 00008904 U.....  
0010 24e8fcff ffffb800 000000c9 c3          $......Contents of section  
.rodata:  
  
0000 48656c6c 6f20576f 726c6421 00  
  
      Hello World!.  
  
Contents of section .comment:  
0000 00474343 3a202855 62756e74 752f4c69 .GCC: (Ubuntu/Li  
0010 6e61726f 20342e34 2e342d31 34756275 naro 4.4.4-14ubu  
0020 6e747535 2920342e 342e3500          ntu5) 4.4.5.  
Disassembly of section .text:00000000 <main>:  
  
0:          55                push    %ebp  
1:          89 e5          mov     %esp,%ebp
```

```

3:          83 e4 f0          and
$0xfffffffff0,%esp
6:          83 ec 10          sub
$0x10,%esp
9:

          b8 00 00 00 00
          mov
          $0x0
,%eax

e:          89 04 24          mov          %eax,
(%esp)
11:

          e8 fc ff ff ff
          call
          12 <main+0x12>

16:          b8 00 00 00 00    mov          $0x0,%eax
1b:          c9                leave
1c:          c3                ret

```

从数据段二进制信息的ASCII形式的显示中，我们看到了汇编语言内定义的字符串数据“Hello World！”。代码段的信息和汇编文件代码信息基本吻合，但是我们发现了很多不同之处。比如汇编文件内的指令“movl\$.LC0, %eax”中的符号.LC0的地址（字符串“Hello World！”的地址）被换成了0。指令“call printf”内符号printf的相对地址被换成了0xffffffffc，即call指令操作数部分的起始地址。

这些区别本质来源于汇编语言符号的引用问题。由于汇编器在处理当前文件的过程中无法获悉符号的虚拟地址，因此临时将这些符号地址设置为默认值0，真正的符号地址只有在链接的时候才能确定。

1.3.4 链接

使用GCC命令进行目标文件链接很简单：

```
gcc hello.o -  
o hello
```

GCC默认使用动态链接，如果要进行静态链接，需加上-static选项：

```
gcc hello.o -  
o hello -  
static
```

这样生成的可执行文件hello便能正常执行了。

我们使用objdump命令查看一下静态链接后的可执行文件内的信息。由于可执行文件中包含了大量的C语言库文件，因此这里不便将文件的所有信息展示出来，仅显示最终main函数的可执行代码。

```
080482c0 <main>:  
  
80482c0:      55                push    %ebp  
80482c1:      89 e5            mov     %esp,%ebp  
80482c3:      83 e4 f0        and     $0xfffffffff0,%esp  
80482c6:      83 ec 10        sub     $0x10,%esp  
80482c9:      b8 28 e8 0a 08
```

	mov		
	\$0x80ae828,		
%eax			
80482ce:	89 04 24	mov	%eax,
(%esp)80482d1:			
	e8 fa 0a 00 00		
	call		
	8048dd0 <_IO_printf>		
80482d6:	b8 00 00 00 00	mov	\$0x0,%eax
80482db:	c9	leave	
80482dc:	c3	ret	

从main函数的可执行代码中，我们发现汇编过程中描述的无法确定的符号地址信息在这里都被修正为实际的符号地址。如“Hello World！”字符串的地址为0x080ae828，printf函数的地址为0x08048dd0。这里符号_IO_printf与printf完全等价，call指令内部相对地址为0x000afa，正好是printf地址相对于call指令下条指令起始地址0x080482d6的偏移。

1.4 设计自己的编译系统

根据以上描述，我们意欲构造一个能将高级语言转化为可执行文件的编译系统。高级语言语法由我们自己定义，它可以是C语言语法，也可以是它的一个子集，但是无论如何，该高级语言由我们根据编程需要自行设计。另外，我们要求生成的可执行文件能正常执行，无论它是Linux系统的ELF可执行文件，还是Windows系统的PE文件，而本书选择生成Linux系统的ELF可执行文件。正如本章开始所描述的，我们要做的就是：自己动手完成当初单击“编译”按钮时计算机在背后做的事情。

然而在真正开工之前，我们需要承认一个事实——我们是无法实现一个像GCC那样完善的工业化编译器的。因此必须降低编译系统实现的复杂度，确保实际的工作在可控的范围内。本书对编译系统的实现做了如下修改和限制：

- 1) 预编译的处理。如前所述，预编译作为编译前期的工作，其主要的目的在于宏命令的展开和文本替换。本质上，预编译器也需要识别源代码语义，它与编译器实现的内容十分相似。通过后面章节对编译器实现原理的介绍，我们也能学会如何构造一个简单的预编译器。因此，在高级语言的文法设计中，本书未提供与预编译处理相关的语

法，而是直接对源代码进行编译，这样使得我们的精力更关注于编译器的实现细节上。

2) 一遍编译的方式。编译器的设计中可以对编译器的每个模块独立设计，比如词法分析器、语法分析器、中间代码优化器等。这样做可能需要对源代码进行多遍的扫描，虽然编译效率相对较低，但是获得的源码语义信息更完善。我们设计的编译系统目标非常直接——保证编译系统输出正确的可执行文件即可，因此采用一遍编译的方式会更高效。

3) 高级语言语法。为了方便大多数读者对文法分析的理解，我们参考C语言的语法格式设计自己的高级语言。不完全实现C语言的所有语法，不仅可以减少重复的工作量，还能将精力重点放在编译算法的实现上，而不是复杂的语言语法上。因此在C语言的基础上，我们删除了浮点类型和**struct**类型，并将数组和指针的维数简化到一维。

4) 编译优化算法。编译器内引入了编译优化相关的内容，考虑到编译优化算法的多样性，我们挑选了若干经典的编译优化算法作为优化器的实现。通过对数据流问题优化算法的实现，可以帮助理解优化器的工作原理，对以后深入学习编译优化算法具有引导意义。

5) 汇编语言的处理。本书的编译器产生的汇编指令属于Intel x86处理器指令集的子集，虽然这间接降低了汇编器实现的复杂度，但是

不会影响汇编器关键流程的实现。另外，编译器在产生汇编代码之前已经分析了源程序的正确性，生成的汇编代码都是合法的汇编指令，因此在汇编器的实现过程中不需要考虑汇编语言的词法、语法和语义错误的情况。

6) 静态链接方式。本书的编译系统实现的链接器采用静态链接的方式。这是因为动态链接器的实现相对复杂，而且其与静态链接器处理的核心问题基本相同。读者在理解了静态链接器的构造的基础上，通过进一步的学习也可以实现一个动态链接器。

7) ELF文件信息。除了ELF文件必需的段和数据，我们把代码全部存放在“`.text`”段，数据存储在“`.data`”段。按照这样的文件结构组织方式，不仅能保证二进制代码正常执行，也有助于我们更好地理解ELF文件的结构和组织。

综上所述，我们所做的限制并没有删除编译系统关键的流程。按照这样的设计，是可以允许一个人独立完成一个较为完善的编译系统的。

1.5 本章小结

本章从编程最基本的话题聊起，描述了初学者接触程序时可能遇到的疑惑，并从编程实践经验中探索代码背后的处理机制。然后，使用最简单的“Hello World！”程序展现主流编译器GCC对代码的处理流程。最后，我们在工业化编译系统的基础上做了一定的限制，提出了本书编译系统需要实现的功能。在接下来的章节中，会对本书中编译系统的设计和实现细节详细阐述。

第2章 编译系统设计

麻雀虽小，五脏俱全。

——《围城》

一个完善的工业化编译系统是非常复杂的，为了清晰地描述它的结构，理解编译系统的基本流程，不得不对它进行“大刀阔斧”地删减。这为自己动手实现一个简单但基本功能完整的编译系统提供了可能。虽然本书设计的是简化后的编译系统，但保留了编译系统的关键流程。正所谓“麻雀虽小，五脏俱全”，本章从全局的角度描述了编译系统的基本结构，并按照编译、汇编和链接的流程来介绍其设计。

2.1 编译程序的设计

编译器是编译系统的核心，主要负责解析源程序的语义，生成目标机器代码。一般情况下，编译流程包含词法分析、语法分析、语义分析和代码生成四个阶段。符号表管理和错误处理贯穿于整个编译流程。如果编译器支持代码优化，那么还需要优化器模块。

图2-1展示了本书设计的优化编译器的结构，下面分别对上述模块的实现方案做简单介绍。

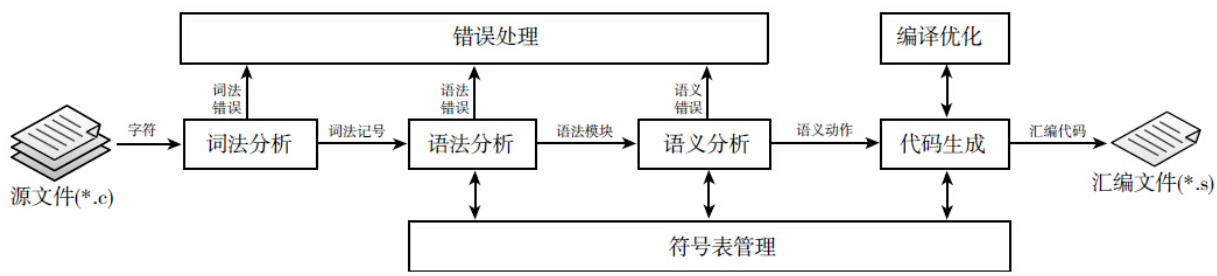


图2-1 编译器结构

2.1.1 词法分析

编译器工作之前，需要将用高级语言书写的源程序作为输入。为了便于理解，我们使用C语言的一个子集定义高级语言，本书后续章节的例子都会使用C语言的一些基本语法作为示例。现在假定我们拥有一段使用C语言书写的源程序，词法分析器通过对源文件的扫描获得高级语言定义的词法记号。所谓词法记号（也称为终结符），反映在高级语言语法中就是对应的标识符、关键字、常量，以及运算符、逗号、分号等界符。见图2-2。

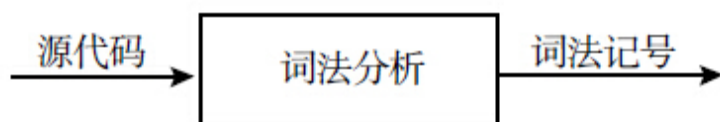


图2-2 词法分析功能

例如语句：

```
var2=var1+100;
```

该语句包含了6个词法记号，它们分别是：“var2”“=”“var1”“+”“100”和分号。

对词法分析器的要求是能正常识别出这些不同形式的词法记号。词法分析器的输入是源代码文本文件内一长串的文本内容，那么如何

从文本串中分析出每个词法记号呢？为了解决这个问题，需要引入有限自动机的概念。

有限自动机能解析并识别词法记号，比如识别标识符的有限自动机、识别常量的有限自动机等。有限自动机从开始状态启动，读入一个字符作为输入，并根据该字符选择进入下一个状态。继续读入新的字符，直到遇到结束状态为止，读入的所有字符序列便是有限自动机识别的词法记号。

图2-3描述了识别标识符的有限自动机。C语言标识符的定义是：一个不以数字开始的由下划线、数字、字母组成的非空字符串。图中的自动机从0号状态开始，读入一个下划线或者字母进入状态1，状态1可以接受任意数量的下划线、字母和数字，同时状态1也是结束状态，一旦它读入了其他异常字符便停止自动机的识别，这样就可以识别任意一个合法的标识符。如果在非结束状态读入了异常的字符，意味着发生了词法错误，自动机停止（当然，上述标识符的有限自动机不会出现错误的情况）。

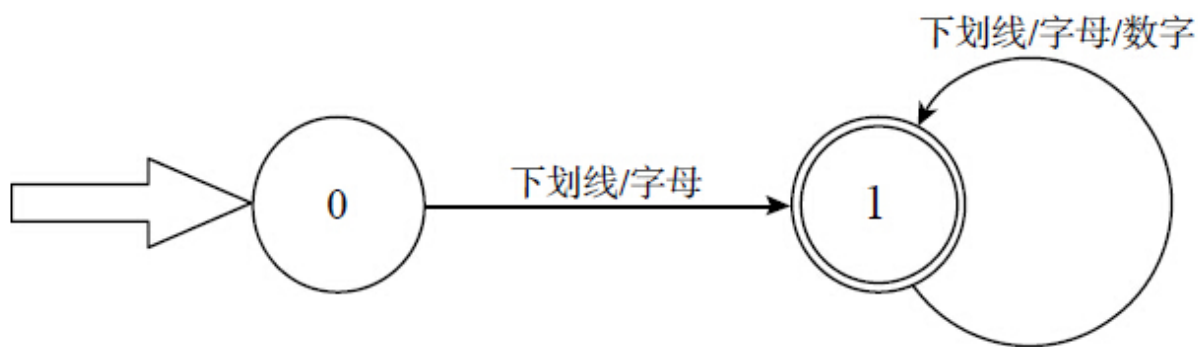


图2-3 标识符有限自动机

我们以赋值语句“var2=var1+100; ”中的变量var2为例来说明有限自动机识别词法记号的工作过程。

识别var2的自动机状态序列和读入字符的对应关系如表2-1所示，结束状态之前识别的字符序列即为合法的标识符。

表2-1 自动机状态序列

	读入	v	a	r	2	=
状态	0	1	1	1	1	结束

使用有限自动机，可以识别出自定义语言包含的所有词法记号。把这些词法记号记录下来，作为下一步语法分析的输入。如果使用一遍编译方式，就不用记录这些词法记号，而是直接将识别的词法记号送入语法分析器进行处理。

2.1.2 语法分析

词法分析器的输入是文本字符串，语法分析器的输入则是词法分析器识别的词法记号序列。语法分析器的输出不再是一串线性符号序列，而是一种树形的数据结构，通常称之为抽象语法树。见图2-4。



图2-4 语法分析功能

继续前面赋值语句的例子，我们可以先看看它可能对应的抽象语法树，如图2-5所示。

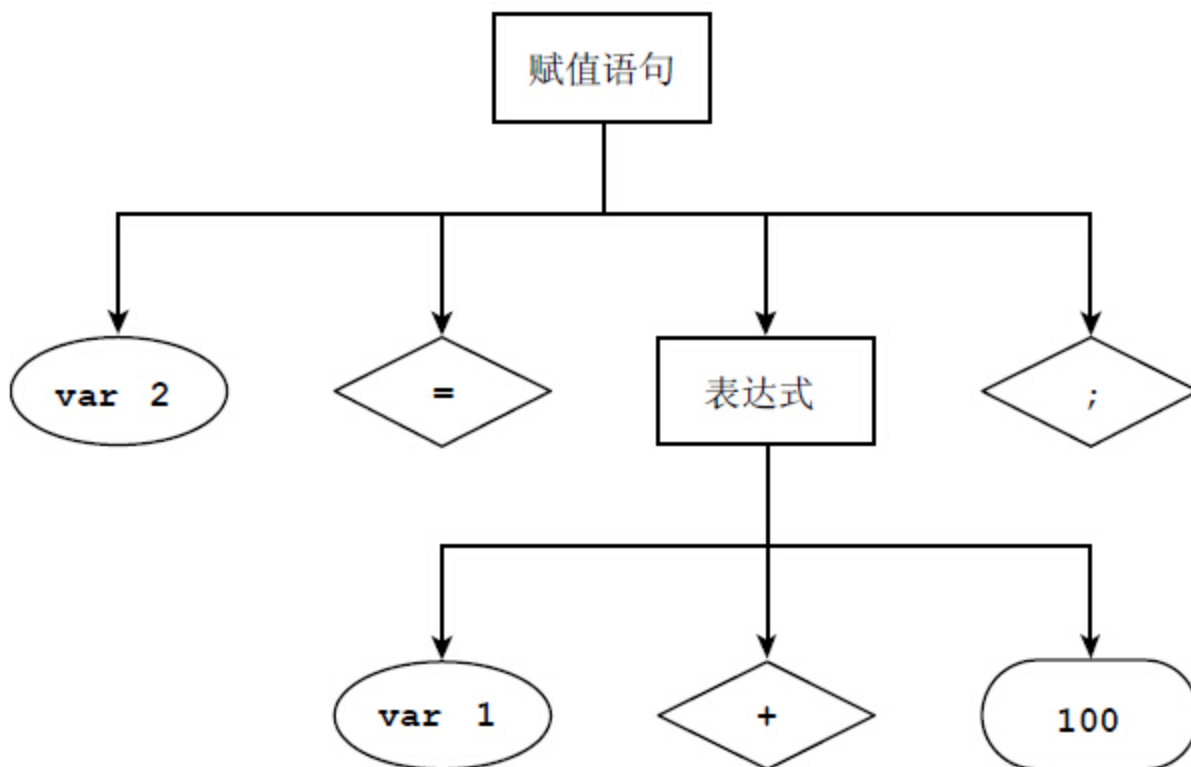


图2-5 抽象语法树示例

从图2-5中可以看出，所有的词法记号都出现在树的叶子节点上，我们称这样的叶子节点为终结符。而所有的非叶子节点，都是对一串词法记号的抽象概括，我们称之为非终结符，可以将非终结符看作一个单独的语法模块（抽象语法子树）。其实，整个源程序是一棵完整的抽象语法树，它由一系列语法模块按照树结构组织起来。语法分析器就是要获得源程序的抽象语法树表示，这样才能让编译器具体识别每个语法模块的含义，分析出程序的整体含义。

在介绍语法分析器的工作之前，需要先获得高级语言语法的形式化表示，即文法。文法定义了源程序代码的书写规则，同时也是语法

分析器构造抽象语法树的规则。如果要定义赋值语句的文法，一般可以表达成如下产生式的形式：

<赋值语句>=>标识符等号<表达式>分号

被“<>”括起来的内容表示非终结符，终结符直接书写即可，上式可以读作“赋值语句推导出标识符、等号、表达式和分号”。显然，表达式也有相关的文法定义。根据定义好的高级语言特性，可以设计出相应的高级语言的文法，使用文法可以准确地表达高级语言的语法规则。

有了高级语言的文法表示，就可以构造语法分析器来生成抽象语法树。在编译原理教材中，描述了很多的文法分析算法，有自顶向下的LL（1）分析，也有自底向上的算符优先分析、LR分析等。其中最常使用的是LL（1）和LR分析。相比而言，LR分析器能力更强，但是分析器设计比较复杂，不适合手工构造。我们设计的高级语言文法，只要稍加约束便能使LL（1）分析器正常工作，因此本书采用LL（1）分析器来完成语法分析的工作。递归下降子程序作为LL（1）算法的一种便捷的实现方式，非常适合手工实现语法分析器。

递归下降子程序的基本原则是：将产生式左侧的非终结符转化为函数定义，将产生式右侧的非终结符转化为函数调用，将终结符转化

为词法记号匹配。例如前面提到的赋值语句对应的子程序的伪代码大致是这样的。

```
void 赋值语句
{
    match(标识符
);
    match(等号
);
    表达式
();
    match(分号
);
}
```

每次对子程序的调用，就是按照前序的方式对该抽象语法子树的一次构造。例如在构造赋值语句子树时，会先构造“赋值语句”根节点，然后依次匹配标识符、等号子节点。当遇到下一个非终结符时，会进入对应的“表达式”子程序内继续按照前序方式构造子树的子树。最后匹配当前子程序的最后一个子节点，完成“赋值语句”子树的构造。整个语法分析就是按照这样的方式构造“程序”树的一个过程，一旦在终结符匹配过程中出现读入的词法记号与预期的词法记号不吻合的情况，便会产生语法错误。

在实际语法分析器实现中，并不一定要显式地构造出抽象语法树。递归下降子程序实现的语法分析器，使得抽象语法树的语法模块都蕴含在每次子程序的执行中，即每次子程序的正确执行都表示识别

了对应的语法模块。因此，可以在语法分析子程序中直接进行后续的工作，如语义分析及代码生成。

2.1.3 符号表管理

符号表是记录符号信息的数据结构，它使用按名存取的方式记录与符号相关的所有编译信息。编译器工作时，少不了符号信息的记录和更新。在本书定义的高级语言中，符号存在两种形式：变量和函数。前者是数据的符号化形式，后者是代码的符号化形式。语义分析需要根据符号检测变量使用的合法性，代码生成需要根据符号产生正确的地址，因此，符号信息的准确和完整是进行语义分析和代码生成的前提。见图2-6。

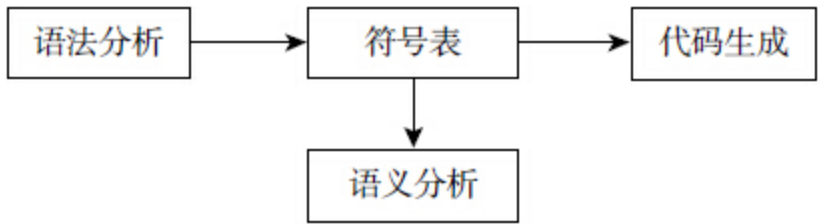


图2-6 符号表管理功能

对于变量符号，需要在符号表中记录变量的名称、类型、区分变量的声明和定义的形式，如果变量是局部变量，还需要记录变量在运行时栈帧中的相对位置。例如以下变量声明语句：

```
extern int var;
```

该语句声明了一个外部的全局变量，记录变量符号的数据结构除了保存变量的名称“**var**”之外，还需要记录变量的类型“**int**”，以及变量是外部变量的声明形式“**extern**”。

对于函数符号，需要在符号表中记录函数的名称、返回类型、参数列表，以及函数内定义的所有局部变量等。例如下面的函数定义代码：

```
int sum(int a,int b)
{
    int c;
    c=a+b;
    return c;
}
```

符号表应该记录函数的返回类型“**int**”、函数名“**sum**”、参数列表“**int, int**”。函数的局部变量除了显式定义的变量“**c**”之外，还暗含参数变量“**a**”和“**b**”。

由于局部变量的存在，符号表必须考虑代码作用域的变化。函数内的局部变量在函数之外是不可见的，因此在代码分析的过程中，符号表需要根据作用域的变化动态维护变量的可见性。

2.1.4 语义分析

编译原理教材中，将语言的文法分为4种：0型、1型、2型、3型，并且这几类文法对语言的描述能力依次减弱。其中，3型文法也称为正规文法，词法分析器中有限自动机能处理的语言文法正是3型文法。2型文法也称为上下文无关文法，也是目前计算机程序语言所采用的文法。顾名思义，程序语言的文法是上下文无关的，即程序代码语句之间在文法层次上是没有关联的。例如在分析赋值语句时，LL（1）分析器无法解决“被赋值的对象是已经声明的标识符吗？”这样的问题，因为语法分析只关心程序语言语法形式的正确性，而不考虑语法模块上下文之间联系的合法性。

然而实际的情况是，程序语言的语句虽然形式上是上下文无关的，但含义上却是上下文相关的。例如：不允许使用一个未声明的变量，不允许函数实参列表和形参列表不一致，不允许对无法默认转换的类型进行赋值和运算，不允许continue语句出现在循环语句之外等，这些要求是语法分析器不能完成的。

根据本书设计的程序语言文法，编译器的语义分析模块（见图2-7）处理如下类似问题：

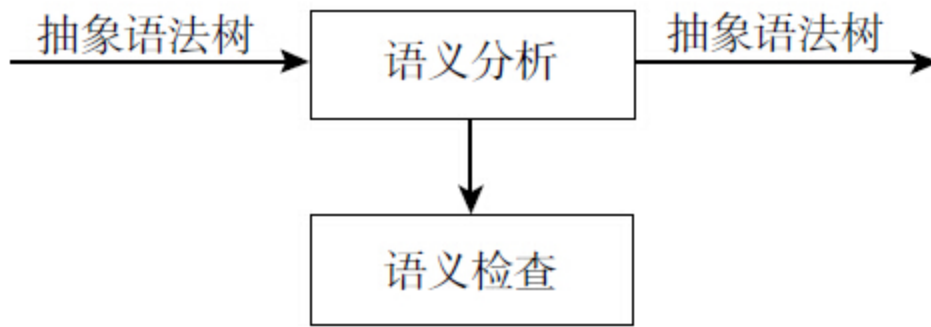


图2-7 语义分析功能

- 1) 变量及函数使用前是否定义?
- 2) **break**语句是否出现在循环或**switch-case**语句内部?
- 3) **continue**语句是否出现在循环内部?
- 4) **return**语句返回值的类型是否与函数返回值类型兼容?
- 5) 函数调用时, 实参列表和形参列表是否兼容?
- 6) 表达式计算及赋值时, 类型是否兼容?

语义分析是编译器处理流程中对源代码正确性的最后一次检查, 只要源代码语义上没有问题, 编译器就可以正常引导目标代码的生成。

2.1.5 代码生成

代码生成是编译器的最后一个处理阶段，它根据识别的语法模块翻译出目标机器的指令，比如汇编语言，这一步称为使用基于语法制导的方式进行代码生成。见图2-8。

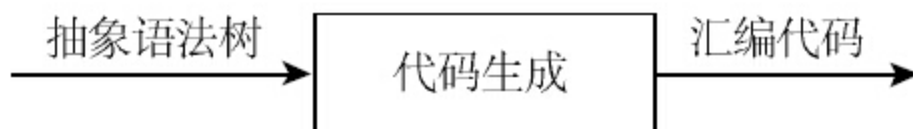


图2-8 代码生成功能

为了便于理解，本书采用常见的Intel格式汇编语言程序作为编译器的输出。继续引用赋值语句“`var2=var1+100;`”作为例子，若将之翻译为汇编代码，其内容可能是：

```
mov eax,[var1]
mov ebx,100
add eax,ebx
mov [tmp],eax
mov eax,[tmp]
mov [var2],eax
```

参考图2-5中的两个非叶子节点，它们分别对应了表达式语法模块和赋值语句语法模块。上面汇编代码的前4行表示将`var1`与100的和存储在临时变量`tmp`中，是对表达式翻译的结果。最后两行表示将临时

变量tmp复制到var2变量中，是对赋值语句的翻译结果。根据自定义语言的语法，需要对如下语法模块进行翻译：

- 1) 表达式的翻译。
- 2) 复合语句的翻译。
- 3) 函数定义与调用的翻译。
- 4) 数据段信息的翻译。

2.1.6 编译优化

现代编译器一般都包含优化器，优化器可以提高生成代码的质量，但会使代码生成过程变得复杂。一般主流的工业化编译器会按照如图2-9所示结构进行设计。

现代编译器设计被分为前端、优化器和后端三大部分，前端包含词法分析、语法分析和语义分析。后端的指令选择、指令调度和寄存器分配实际完成代码生成的工作，而优化器则是对中间代码进行优化操作。实现优化器，必须设计编译器的中间代码表示。中间代码的设计没有固定的标准，一般由编译器设计者自己决定。

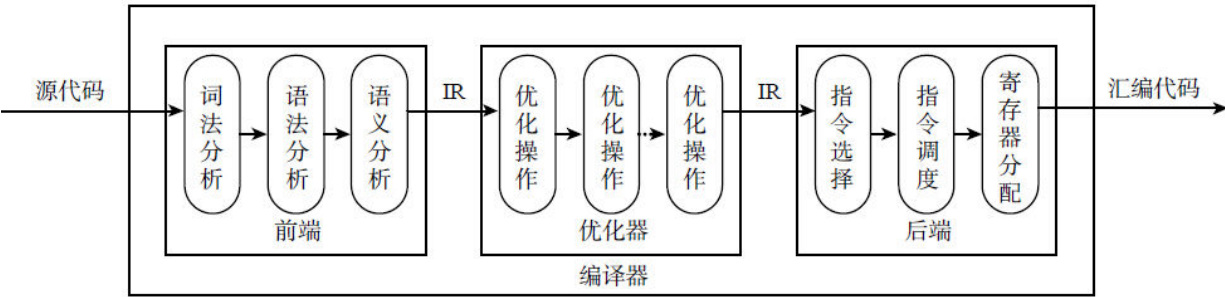


图2-9 现代编译器结构

由于中间代码的存在，使得语法制导翻译的结果不再是目标机器的代码，而是中间代码。按照我们自己设计的中间代码形式，上述例子生成的中间代码可能是如下形式：

```
tmp=var1+100  
var2=tmp
```

即使优化器没有对这段代码进行处理，编译器的后端也能正确地把这段中间代码翻译为目标机制指令。根据指令选择和寄存器分配算法，得到的目标机器指令可能如下：

```
mov eax,[var1]  
add eax,100  
mov [var2],eax
```

编译器后端在指令选择阶段会选择更“合适”的指令实现中间代码的翻译，比如使用“add eax, 100”实现tmp=var1+100的翻译。在寄存器分配阶段会尽可能地将变量保存在寄存器内，比如tmp一直保存在eax中。

中间代码的抽象程度一般介于高级语言和目标机器语言之间。良好的中间代码形式使得中间代码生成、目标代码生成以及优化器的实现更加简单。我们设计的优化器实现了常量传播、冗余消除、复写传播和死代码消除等经典的编译优化算法。先通过一个简单的实例说明中间代码优化的工作。

```
var1=100;  
var2=var1+100;
```

将上述高级语言翻译为中间代码的形式如下：

```
var1=100  
tmp=var1+100
```

```
var2=tmp
```

常量传播优化使编译器在编译期间可以将表达式的结果提前计算出来，因此经过常量传播优化后的中间代码形式如下：

```
var1=100  
tmp=200  
var2=200
```

死代码消除优化会把无效的表达式从中间代码中删除，假如上述代码中只有变量`var2`在之后会被使用，那么`var1`和`tmp`都是无效的计算。因此，消除死代码后，最终的中间代码如下：

```
var2=200
```

再经过后端将之翻译为汇编代码如下：

```
mov [var2],200
```

由于本书篇幅及作者水平所限，在不能实现所有的编译优化算法的情况下，选择若干经典的优化算法来帮助读者理解优化器的基本工作流程。

至此，我们简单介绍了高级语言源文件转化为目标机器的汇编代码的基本流程。本书设计的编译器支持多文件的编译，因此编译器会为每个源文件单独生成一份汇编文件，然后通过汇编器将它们转换为二进制目标文件。汇编过程中涉及目标机器的指令格式和可执行文件

的内容，为了便于理解汇编器的工作流程，需要提前准备与操作系统和硬件相关的知识。

2.2 x86指令格式

编译系统的汇编器需要把编译器生成的汇编语言程序转化为x86格式的二进制机器指令序列，然后将这些二进制信息存储为ELF格式的目标文件。因此需要先了解二进制机器指令的基本结构。

如图2-10所示，在x86的指令结构中，指令被分为前缀、操作码、ModR/M、SIB、偏移量和立即数六个部分。本书设计的编译器生成的汇编指令中不包含前缀，这里暂时不介绍它的含义。操作码部分决定了指令的含义和功能，ModR/M和SIB字节为扩充操作码或者为指令操作数提供各种不同的寻址模式。如果指令含有偏移量和立即数信息，就需要把它们放在指令后边的对应位置。



图2-10 x86指令格式

这里使用一个简单的例子与表2-2说明x86指令结构的含义，例如汇编指令：

`add eax,ebx`

表2-2 二进制指令编码

指令格式	操作码	mod 字段	reg 字段	r/m 字段	指令编码
add r/m32,reg	0x01	11	011	000	0000 0011 1100 0011
add reg,r/m32	0x03	11	000	011	0000 0001 1101 1000

查阅Intel的指令手册，当操作数为32位寄存器时，add指令的操作码是0x01或者0x03，它们对应的指令格式是add r/m32，reg和add reg，r/m32。在ModR/M字节的定义中，高两位mod字段为0b11时表示指令的两个操作数都是寄存器，低三位表示r/m操作数寄存器的编号，中间三位表示reg操作数寄存器的编号。Intel定义eax寄存器编号为0b000，ebx寄存器编号为0b011。如果我们采用操作码0x01，reg应该记录ebx的编号0b011，r/m32记录eax编号0b000，mod字段为0b11。因此该指令的ModR/M字节为：

11 011 000 => 0xd8

同理，若采用操作码0x03的话，ModR/M字节应该是：

11 000 011 => 0xc3

指令不再含有其他信息，因此不存在SIB和偏移量、立即数字段。这样“add eax，ebx”指令就有两种二进制表示形式：0x01d8与0x03c3。

通过这个例子可以得出结论：在汇编器语法分析阶段，应该记录生成的二进制指令需要的信息。指令的名称决定操作码，指令的寻址

方式决定ModR/M和SIB字段，指令中的常量决定偏移量和立即数部分。

由于本书设计的编译器所生成的汇编指令的种类有限，因此降低了汇编器对指令信息分析的复杂度，但是还有大量的其他类型的指令需要具体分析，这些内容会在以后章节中阐述。

2.3 ELF文件格式

ELF文件格式描述了Linux下可执行文件、可重定位目标文件、共享目标文件、核心转储文件的存储格式。本书设计的编译系统只关心可执行文件和可重定位目标文件的格式，如果要设计动态链接器的话，则还需要了解共享目标文件的内容。

ELF文件信息的一般存储形式如图2-11所示。

文件头(ELF Header)	52
程序头表(Program Header Table)	(32*程序头表项个数)
代码段(.text)	
数据段(.data)	
bss 段(.bss)	0
段表字符串表(.shstrtab)	
段表(Section Header Table)	(40*段表项个数)
符号表(.symtab)	(16*符号表项个数)
字符串表(.strtab)	
重定位表(.rel .text)	(8*重定位表项个数)
重定位表(.rel .data)	(8*重定位表项个数)

图2-11 ELF文件

在Linux下，可以使用readelf命令查看ELF文件的信息。如果要查看1.3.3节生成的hello.o的信息，可以使用如下命令查看ELF的所有关键信息：

```
readelf -  
a hello.o
```

在ELF文件中，最开始的52个字节记录ELF文件头部的信息，通过它可以确定ELF文件内程序头表和段表的位置及大小。以下列出了hello.o文件头信息。

```
ELF Header:  
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00  
  Class:                                ELF32  
  Data:                                  2's complement,  
little endian  
  Version:                               1 (current)  
  OS/ABI:                                UNIX - System V  
  ABI Version:                           0  
  Type:                                   REL (Relocatable file)  
  
  Machine:                               Intel 80386  
  Version:                               0x1  
  Entry point address:  
  
                                         0x0  
  
Start of program headers:  
  
                                         0 (bytes into file)  
  
Start of section headers:  
  
                                         224 (bytes into file)  
  
Flags:                                   0x0  
Size of this header:                     52 (bytes)  
Size of program headers:                 0 (bytes)
```

Number of program headers:	0
Size of section headers:	40 (bytes)
Number of section headers:	11
Section header string table index:	8

紧接着文件头便是程序头表，它记录程序运行时操作系统如何将文件加载到内存，因此只有可执行文件包含程序头表。使用readelf查看1.3.4节静态链接生成的hello文件，可以看到它的程序头表，类型为LOAD的表项表示需要加载的段。以下列出它的程序头表信息。

Program Headers:							
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD							
	0x000000						
	0x08048000						
	0x08048000						
	0x84fd2						
	0x84fd2						
R E							
	0x1000						
LOAD							
	0x085f8c						
	0x080cdf8c						
	0x080cdf8c						
	0x007d4						
	0x02388						
RW							
	0x1000						
NOTE	0x0000f4	0x080480f4	0x080480f4	0x00044	0x00044	R	0x4
TLS	0x085f8c	0x080cdf8c	0x080cdf8c	0x00010	0x00028	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4
GNU_RELRO	0x085f8c	0x080cdf8c	0x080cdf8c	0x00074	0x00074	R	0x1

ELF文件最关键的结构是段表，这里的段表示文件内的信息块，与汇编语言内的段并非同一个概念。段表记录了ELF文件内所有段的位置和大小等信息。在所有的段中，有保存代码二进制信息的代码段、存储数据的数据段、保存段表名称的段表字符串表段和存储程序字符串常量的字符串表段。符号表段记录汇编代码中定义的符号信息，重定位表段记录可重定位目标文件中需要重定位的符号信息。

hello.o的段表如下：

Section Headers:							
[Nr]	Name		Type	Addr	Off	Size	ES
Lk	Inf	Al					Flg
[0]			NULL	00000000	000000	000000	00
0	0	0					
[1]							
	.text						
		PROGBITS					
	00000000						
	000034						
	00001d						
	00						
	AX						
	0						
	0						
	4						
[2]							
	.rel.text						
		REL					
	00000000						
	000350						
	000010						

```

08
    9
1
4

[3]
.data
    PROGBITS
    00000000
    000054
    000000
    00
    WA
    0
    0
    4

[4] .bss      NOBITS    00000000  000054  000000    00    WA
0   0         4
[5] .rodata   PROGBITS  00000000  000054  00000d    00    A
0   0         1
[6] .comment   PROGBITS  00000000  000061  00002c    01    MS
0   0         1
[7] .note.GNU-stack PROGBITS  00000000  00008d  000000    00
0   0         1
[8] .shstrtab  STRTAB    00000000  00008d  000051    00
0   0         1
[9]
.symtab
    SYMTAB
    00000000
    000298
    0000a0
    10
    10
    8
    4

```

[10].strtab	STRTAB	00000000	000338	000015	00
0	0	1			

符号表段是按照表格形式存储符号信息的，我们可以看到主函数和printf函数的符号项。

Symbol table '.symtab' contains 10 entries:						
Num:	Value	Size	Type	Bind	Vis	Ndx
Name						
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND
1:	00000000	0	FILE	LOCAL	DEFAULT	ABS
hello.c						
2:	00000000	0	SECTION	LOCAL	DEFAULT	1
3:	00000000	0	SECTION	LOCAL	DEFAULT	3
4:	00000000	0	SECTION	LOCAL	DEFAULT	4
5:	00000000	0	SECTION	LOCAL	DEFAULT	5
6:	00000000	0	SECTION	LOCAL	DEFAULT	7
7:	00000000	0	SECTION	LOCAL	DEFAULT	6
8:	00000000					
29						
	FUNC					
	GLOBAL					
	DEFAULT					
	1					
	main					
9:	00000000					
0						
	NOTYPE					
	GLOBAL					
	DEFAULT					
	UND					
	printf					

重定位表也是按照表格形式存储的，很明显，printf作为外部符号是需要重定位的。

```
Relocation section '.rel.text' at offset 0x350 contains 2 entries:
Offset      Info      Type      Sym.Value    Sym.Name
0000000a     00000501   R_386_32   00000000     .rodata00000012

      00000902

      R_386_PC32

      00000000

      printf
```

从ELF文件格式的设计中可以看出，可执行文件其实就是按照一定标准将二进制数据和代码等信息包装起来，方便操作系统进行管理和使用。从文件头可以找到程序头表和段表，从段表可以找到其他所有的段。因此，在汇编语言输出目标文件的时候，就需要收集这些段的信息，并按照ELF格式组装目标文件。这样做不仅有利于使用操作系统现有的工具调试文件信息，也为后期链接器的实现提供了方便。

另外需要说明的是，对于ELF文件格式的定义，Linux提供了头文件描述。在系统目录/usr/include/elf.h提供的elf.h头文件中描述了标准ELF文件的数据结构的定义，在实现汇编器和链接器的代码中都使用了该头文件。

2.4 汇编程序的设计

通过对汇编器已有的了解，可以发现汇编器和编译器的实现非常相似。编译器是将高级语言翻译为汇编语言的转换程序，汇编器则是将汇编语言翻译为目标机器二进制代码的转换程序。汇编器实际就是汇编语言的“编译器”，虽然汇编语言并非高级语言。

汇编器也包含词法分析、语法分析、语义处理、代码生成四个基本流程。但前面讨论过，本书设计的汇编器面向编译器所生成的汇编代码，汇编代码的正确性由编译器保证，因此汇编器不需要进行错误检查以及语义的正确性检查。本书设计的汇编器结构如图2-12所示。

相比于编译器，汇编器的工作重点放在目标文件信息的收集和二进制指令的生成上。下面分别介绍汇编器的基本模块。

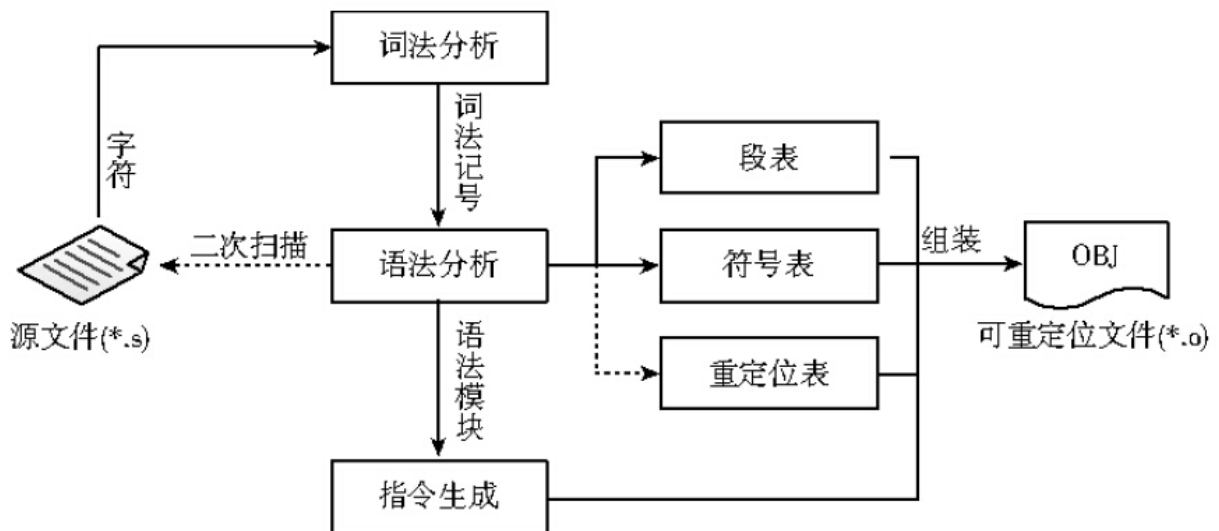


图2-12 汇编器结构

2.4.1 汇编词法、语法分析

汇编语言有独立的词法记号，对于汇编词法的分析，只需要构造相应的词法有限自动机就可以了。举一个简单的例子：

```
mov eax,[ebp-8]
```

该指令有8个词法记号，它们分别是：‘mov’ ‘eax’ 逗号‘[’ ‘ebp’ ‘-’ ‘8’ 和‘]’。汇编器的词法分析器将词法记号送到语法分析器用于识别汇编语言的语法模块。同样，我们需要构造汇编语言语法分析器，在这里可以提前看一下上述汇编指令的抽象语法树，如图2-13所示。

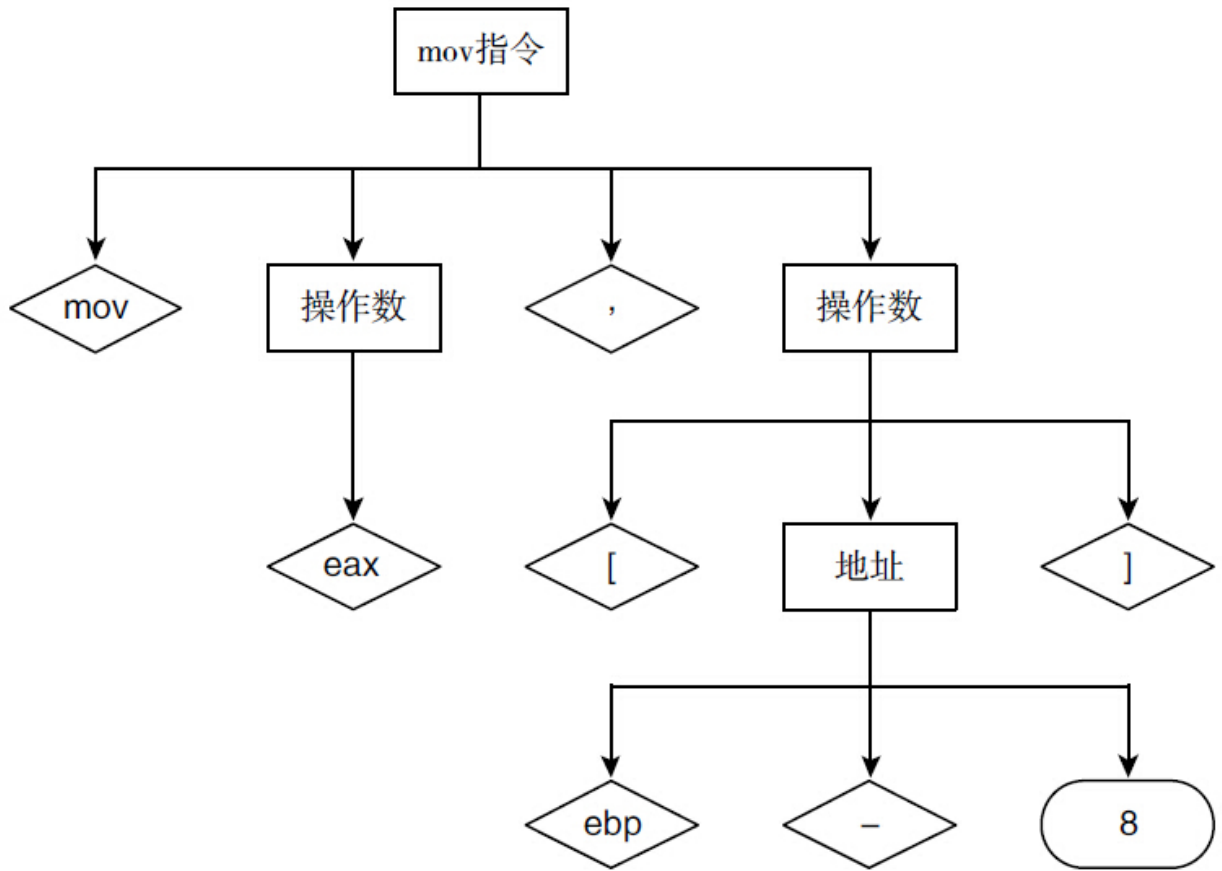


图2-13 汇编指令抽象语法子树

图2-13中是简化后的抽象语法树，与编译器类似，语法分析器会在非叶子节点处识别语法模块，以产生语义动作。由于汇编器要输出可重定位目标文件，因此在语法分析时要收集目标文件的相关信息。比如记录代码段和数据段的长度、目标文件符号表的内容、重定位表的内容等，这些操作都在语法分析器识别每个语法模块时使用语法制导的方式完成。

另外，汇编器和编译器最大的不同是汇编器需要对源文件进行两遍扫描，其根本原因是汇编语言允许符号的后置定义，例如汇编语言

常见的跳转指令：

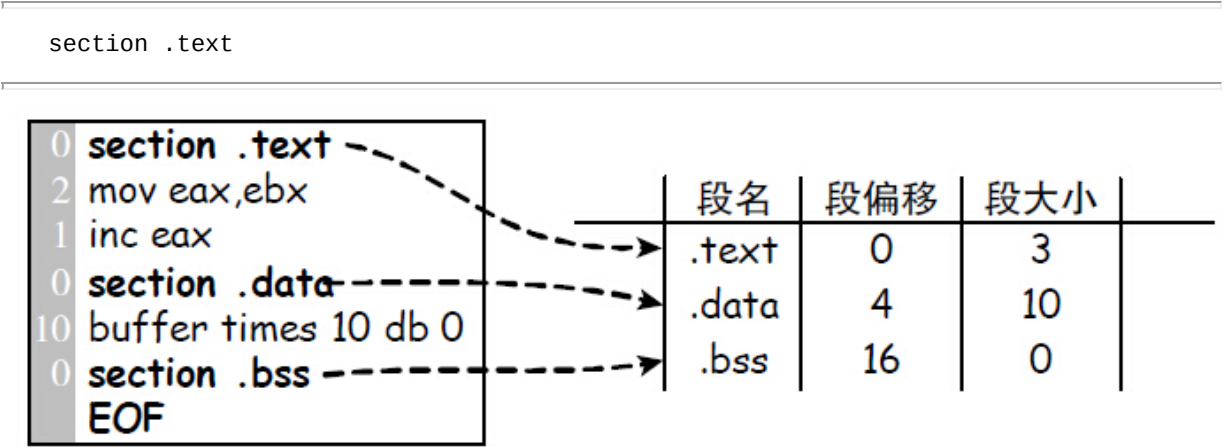
```
    jmp L
L:
```

很明显，在第一遍分析`jmp`指令的时候，汇编器并不知道符号`L`是否已经定义。因此，汇编器需要通过第一遍扫描获取符号的信息，在第二遍扫描时使用符号的信息。

2.4.2 表信息生成

汇编器的符号表除了记录符号的信息之外，还需要记录段相关的信息以及重定位符号的信息，这些信息都是生成可重定位目标文件所必需的。

对于段表的信息，可以在汇编器识别section语法模块时进行处理。比如声明代码段的汇编代码及段表信息生成（见图2-14）。



导出的符号信息，如图2-15所示。最明显的一个例子就是使用**equ**命令定义的符号，这个符号对汇编器来说是一个符号，但在ELF文件内，它就是一个数字常量，不存在符号信息。

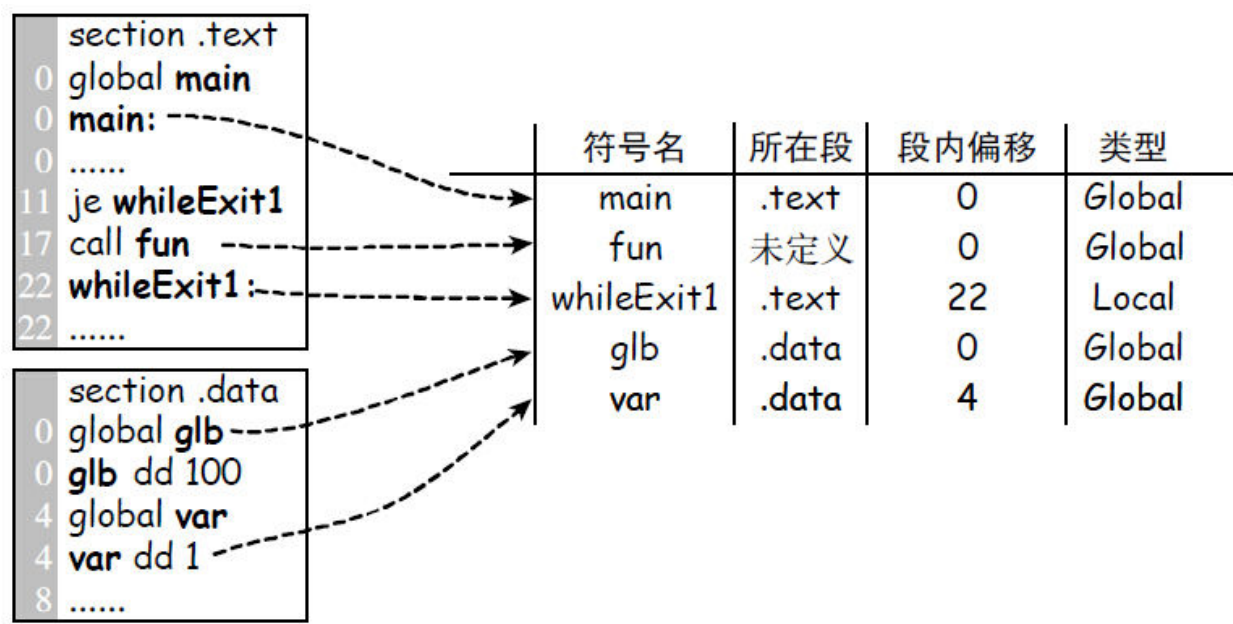


图2-15 符号表信息生成

目标文件链接时会重新组织代码段、数据段的位置。这样段内定义的所有符号的地址以及引用符号的数据和指令都会产生偏差，这时就重新计算符号的地址，修改原来的地址，也就是常说的重定位。重定位一般分为两大类：绝对地址重定位和相对地址重定位。在重定位表内，需要记录符号重定位相关的所有信息（见图2-16）。

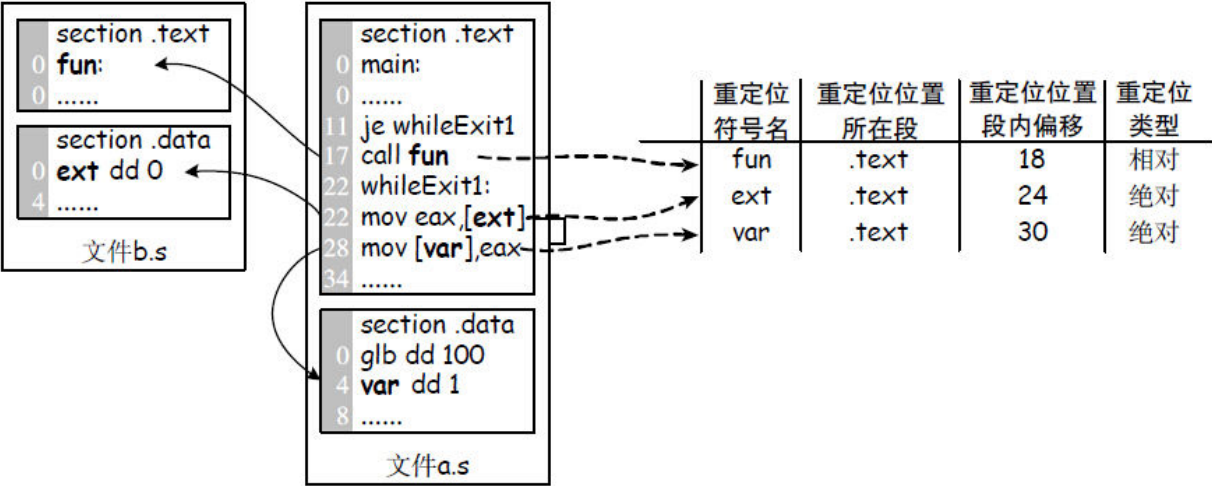


图2-16 重定位表信息生成

2.4.3 指令生成

2.2节介绍了x86指令的基本结构。同样，在汇编器语法分析时，需要根据指令的语法模块收集这些指令的结构信息。比如操作码、**ModR/M**字段、**SIB**字段、偏移量、立即数，然后按照指令的结构将上述信息写入文件即可。

首先，指令名和操作码一般是一对多的关系，因此需要根据具体的操作数类型或长度来决定操作码的值。按照操作数不同建立一张指令的操作码表来执行操作码的查询是一种有效的解决方案。

其次，有些指令的**ModR/M**字段的**reg**部分与操作码有关，但不需要输出**ModR/M**字段，汇编器需要单独处理这些特殊的指令操作码。另外，**ModR/M**字段中包含是否扩展了**SIB**字段的信息。

除了正确输出指令的二进制信息外，汇编器在遇到对符号引用的指令时还要记录相关重定位信息，比如重定位地址、重定位符号、重定位类型等。

最后，参考之前介绍的**ELF**文件结构，汇编器将收集到的段信息和二进制数据组装到目标文件内。

至此，根据已描述的汇编器主要工作流程，可以生成标准的**ELF**可重定位目标文件。那么，如何把这些分散的目标文件合并成我们最终想要的可执行文件，便是接下来要介绍的链接器的工作内容。

2.5 链接程序的设计

本书欲设计一个简洁的静态链接器，以满足上述汇编器产生的目标文件的链接需求。它的工作内容是把多个可重定位目标文件正确地合并为可执行文件，但链接器不是对文件进行简单的物理合并。除了合并同类的段外，链接器需要为段分配合适的地址空间，还需要分析目标文件符号定义的完整性，同时对符号的地址信息进行解析，最后还有链接器最关键的工作——重定位。本书设计的链接器结构如图2-17所示。

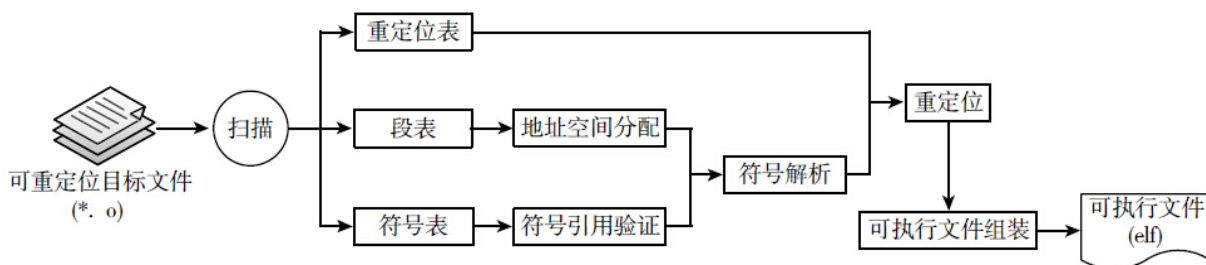


图2-17 链接器结构

段的地址空间分配除了为加载器提供相应的信息外，还可为段内符号地址的计算提供依据。符号解析除了验证符号引用和定义的正确性之外，还需要计算出每个符号表内符号的虚拟地址。重定位可以让每个引用符号的数据或者代码具有正确的内容，以保证程序的正确性，下面就按照这三个方面分别介绍链接器的工作。

2.5.1 地址空间分配

在汇编器生成的目标文件内，是无法确定数据段和代码段的虚拟地址的，因此将它们的段地址都设置为0。链接器是这些代码和数据加载到内存执行之前的最后一道处理，因此要为它们分配段的基址。

链接器按照目标文件的输入顺序扫描文件信息，从每个文件的段表中提取出各个文件的代码段和数据段的信息。假设可执行文件段加载后的起始地址是0x080408000，链接器从该地址开始，就像“摆积木”似的将所有文件的同类型段合并，按照代码段、数据段、“.bss”段的顺序依次决定每个段的起始地址，此时需要考虑段间对齐产生的偏移以及特殊的地址计算方式（参考第5章关于程序头表的描述）。

图2-18展示了链接器将目标文件a.o和b.o链接为可执行文件ab时，地址空间分配的效果。a.o的数据段大小为0x08字节，代码段大小为0x4a字节；b.o的数据段大小为0x04字节，代码段大小为0x21字节。链接后的可执行文件ab的数据段大小为0x0c字节，代码段大小为0x6d字节（对齐b.o的代码段消耗2字节）。代码段的起始地址为0x08048080，结束地址为0x08048080+0x6d=0x080480ed。数据段起始地址为0x080490f0，结束地址为0x080490f0+0x0c=0x080490fc。

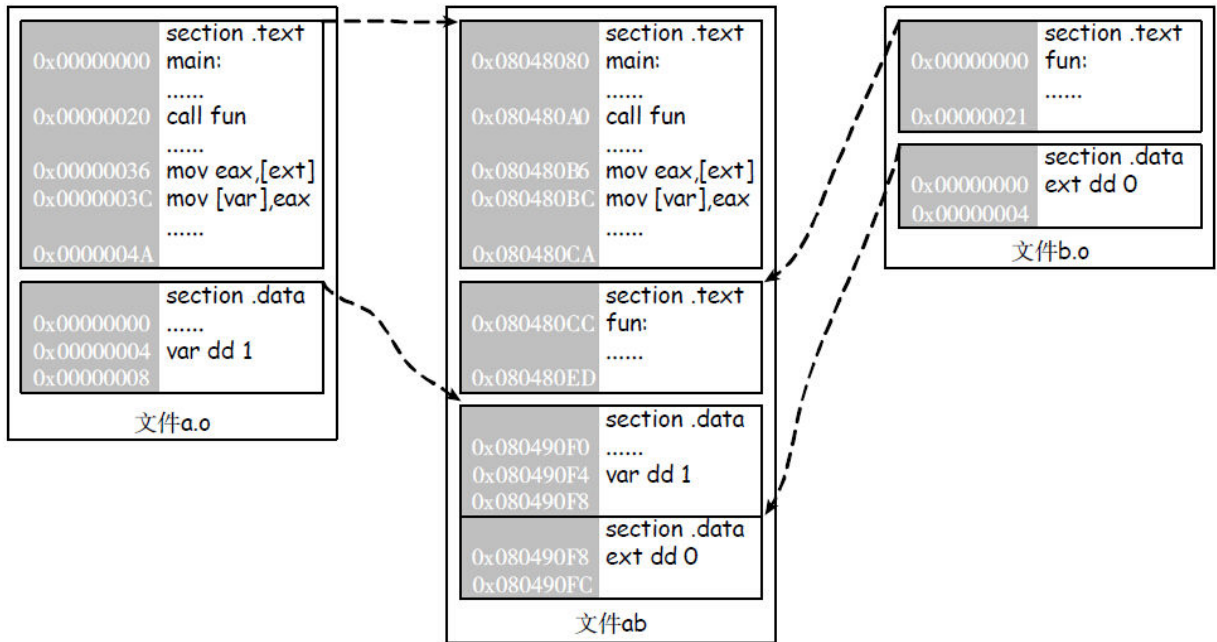


图2-18 地址空间分配

2.5.2 符号解析

如果说地址空间分配是为段指定地址的话，那么符号解析就是为段内的符号指定地址。对于一个汇编文件来说，它内部使用的符号分为两类：一类来自自身定义的符号，称为内部符号。内部符号在其段内的偏移是确定的，当段的起始地址指定完毕后，内部符号的地址按照如下方式计算：

$\text{符号地址} = \text{符号所在段基址} + \text{符号所在段内偏移}$

另一类来自其他文件定义的符号，本地文件只是使用该符号，这类符号称为外部符号。外部符号地址在本地文件内是无法确定的，但是外部符号总定义在其他文件中。外部符号相对于定义它的文件就是内部符号了，同样使用前面的方式计算出它的地址，而使用该符号的本地文件需要的也是这个地址。

在重定位目标文件内，符号表记录了符号的所有信息。对于本地定义的符号，符号表项记录符号的段内偏移地址。对于外部引用的符号，符号表项标识该符号为“未定义的”。当链接器扫描到定义该外部符号的目标文件时，就需要将该外部符号的地址修改为正确的符号地址。最终的结果使得所有目标文件内的符号信息，无论是本地定义的还是外部定义的都是完整的、正确的。

链接器在扫描重定位目标文件的符号表时会动态地维护两个符号集合。一个记录所有文件定义的全局符号集合**Export**，该集合内的所有符号允许被其他文件引用。还有一个记录所有文件使用的未定义符号的集合**Import**，该集合内所有符号都来源于其他目标文件。文件扫描完毕后，链接器需要验证**Import**集合是否是**Export**的子集。如果不是，就表明存在未定义的符号。未定义的符号信息是未知的，链接器无法进行后续的操作，因而会报错。如果验证成功，则表明所有文件引用的外部符号都已定义，链接器才会将已定义的符号信息拷贝到未定义符号的符号表项。

符号解析完毕后，所有目标文件符号表内的所有符号都获得了完整、正确的符号地址信息。比如图2-18内的符号**var**、**ext**和**fun**在符号解析后的符号地址分别为0x080490f4、0x080490f8和0x080480cc。

2.5.3 重定位

重定位从本质上来说就是地址修正。由于目标文件在链接之前不能获取自己所使用符号的虚拟地址信息，因此导致依赖于这些符号的数据定义或者指令信息缺失。汇编器在生成目标文件的时候就记录下所有需要重定位的信息。链接器获取这些重定位信息，并按照重定位信息的含义修改已经生成的代码，使得最终的代码正确、完整。

之所以称重定位是链接器最关键的操作，主要是因为地址空间分配和符号解析都是为重定位做准备的。程序在运行时，段的信息、符号的信息都显得“微不足道”，因为CPU只关心文件内的代码和数据。即便如此，也不能忽略地址空间分配和符号解析的重要性。既然重定位是对已有二进制信息的修改，因此作为链接器需要清楚几件事情：

- 1) 在哪里修改二进制信息？
- 2) 用什么信息进行修改？
- 3) 按照怎样的方式修改？

这三个问题反映在重定位中对应的三个参数：重定位地址、重定位符号和重定位类型。

重定位地址在重定位表中没有直接记录，因为在重定位目标文件内，段地址还没确定下来，它只记录了重定位位置所在段内的偏移，在地址空间分配结束后，我们使用如下公式计算出重定位地址：

重定位地址=重定位位置所在段基址+重定位位置的段内偏移

重定位符号记录着被指令或者数据使用的符号信息，比如**call**指令的标号、**mov**指令使用的变量符号等。在符号解析结束后，重定位符号的地址就已经确定了。

重定位类型决定修改二进制信息的方式，即绝对地址重定位和相对地址重定位。

在确定了重定位符号地址和重定位地址后，根据重定位的类型，链接器便可以正确修改重定位地址处的符号地址信息。

至此，链接器的主要工作流程描述完毕。作为编译系统的最后一个功能模块，链接器与操作系统的关系是最密切的，比如它需要考虑页面地址对齐、指令系统结构以及加载器工作的特点等。

2.6 本章小结

本章介绍了编译系统的设计，并按照编译、汇编和链接的顺序阐述了它们的内部实现。同时，也介绍了x86指令和ELF文件结构等与操作系统及硬件相关的知识。

通过以上的描述，可以了解高级语言如何被一步步转化为汇编语言，以及词法分析、语法分析、语义分析、符号表和代码生成作为编译器的主要模块，其内部是如何实现的。汇编器在把汇编语言程序转化为二进制机器代码时，做了怎样的工作；汇编器的词法和语法分析与编译器有何不同；汇编器如何生成二进制指令和目标文件的信息。链接器在处理目标文件时是如何进行地址分配、符号解析以及重定位的，它生成的可执行文件和目标文件有何不同等。

通过对这些问题的简要描述，我们对编译系统的工作流程有了全局的认识。至于具体的实现细节会在以后的章节中以一个自己动手实现的编译系统为例详细进行介绍，下面就让我们开始实现一个真正的编译系统吧！

第3章 编译器构造

千举万变，其道一也。

——《荀子》

从本章开始，将详细阐述如何构造一个自定义语言的编译器。在实现编译器之前，必须弄清编译器要处理什么样的语言。本书设计的自定义语言是C语言的子集。C语言本身容易理解，为多数人熟知，了解C语言编译器的实现过程对加深理解C语言的本质大有裨益。此外，选用C语言的子集可以有效降低编译器实现的复杂度，将我们的精力重点放在编译器实现的流程，而非繁杂、重复的编码工作上。

参考图2-1描述的编译器的结构，接下来详细阐述编译器每个功能模块的实现。

3.1 词法分析

词法分析是编译器处理流程中的第一步，它顺序扫描源文件内的字符，通过与词法记号的有限自动机进行匹配，产生各式各样的词法记号。图3-1描述了词法分析器的结构，将从源文件内按序扫描字符的功能独立出来，称之为扫描器，而与有限自动机进行匹配产生词法记号的功能称为解析器。

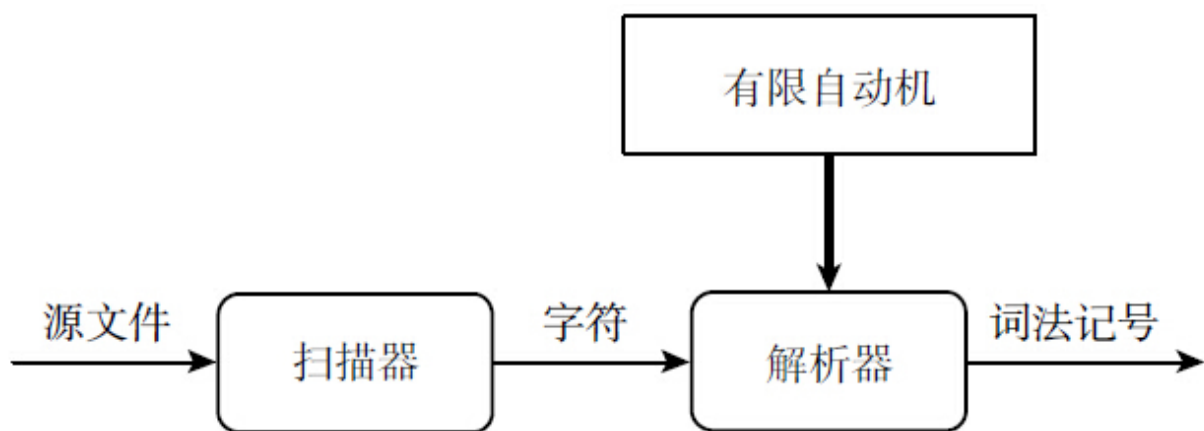


图3-1 词法分析器结构

根据词法分析器的结构，需要解决以下四个问题：

- 1) 扫描器如何实现源文件字符的扫描，它与普通的读文件有什么区别？
- 2) 词法记号是如何定义的？

3) 为什么有限自动机可以识别词法记号?

4) 解析器是如何利用有限自动机将一串字符转化为词法记号的?

带着这四个问题, 我们一起探索词法分析器的实现。

这样做可以实现扫描器的基本功能，但是并不高效。因为词法分析器每次调用`scan`函数获取源文件内的下一个字符时，都会产生一次对磁盘的读操作，而磁盘的I/O操作是比较耗时的。更高效的方式是使用一块缓冲区保存后续的多个字符，每次调用`scan`时首先从缓冲区内按序获取字符，只有缓冲区为空时才会读取磁盘重新加载缓冲区。这有点类似于Linux文件系统的预读机制。

使用缓冲区进行文件扫描的算法流程如图3-3所示。

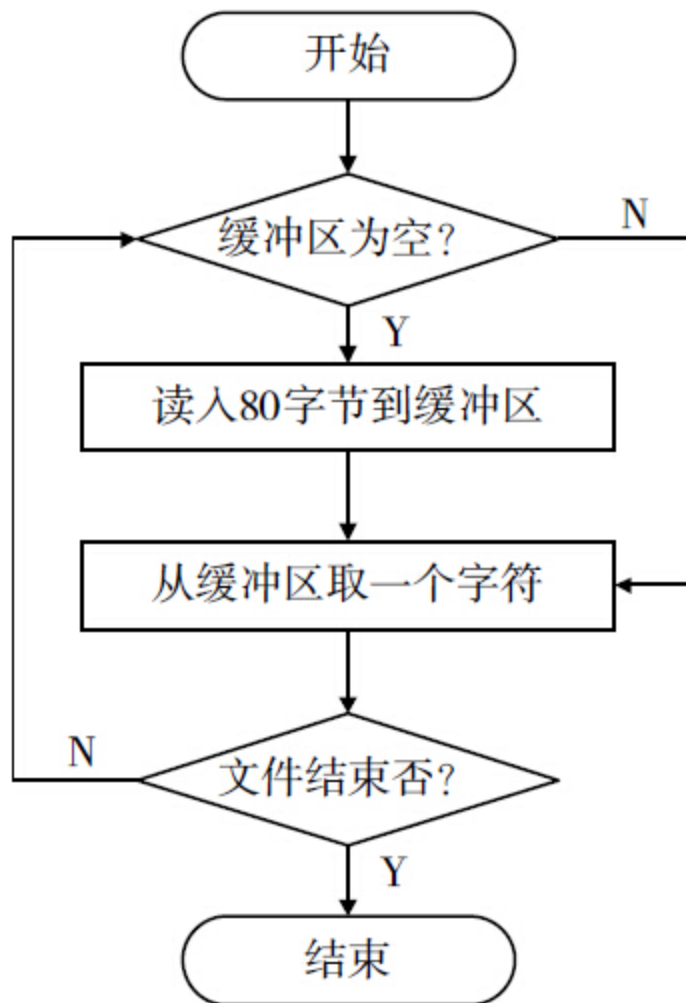


图3-3 扫描器算法

扫描器使用80字节长度的缓冲区，每次从缓冲内读取字符，如果缓冲区为空，则从源文件内加载新的80字节数据到缓冲区，直到文件读取完毕。算法的实现代码如下：

```
1  #define BUFLen 80                                //缓冲区大小
2  int lineLen=0;                                    //缓冲区内的数据长度
3  int readPos=-1;                                   //读取位置
```

```

4 char line[BUFLen]; //缓冲区

5 int lineNum=1; //行号

6 int colNum=0; //列号

7 char lastch=ch; //上一个字符

8 char scan
(FILE*file){
9     if(!file) //没有文件

10         return -1;
11     if(readPos==lineLen-1){ //缓冲区读取完毕

12         lineLen=fread
(line,1,BUFLen,file); //重新加载缓冲区数据

13         if(lineLen==0){ //没有数据了

14             lineLen=1; //数据长度为

15             line[0]=-1; //文件结束标记

16         }
17         readPos=-1; //恢复读取位置

18     }
19     readPos++; //移动读取点

20     char ch=line[readPos]; //获取新的字符

21     if(lastch=='\n'){ //新行

22         lineNum++; //行号累加

23         colNum=0; //列号清空

24     }
25     if(ch==-1){ //文件结束，自动关闭

26         fclose(file);
27         file=NULL;
28     }
29     else if(ch!='\n') //不是换行

```



```
30          colNum++;                                //列号递增

31      lastch=ch;                                    //记录上一个字符

32      return ch;                                    //返回字符

33 }
```

第9行判断文件指针是否为空，如果为空，则直接返回文件结束符-1。

第11行判断读取位置readPos是否到达缓冲区内数据的末尾，如果是，则使用fread库函数重新加载缓冲区，读入BUFLEN字节数据。

第13行判断fread的返回值，如果lineLen等于0，则说明文件结束。此时将lineLen设置为1，并在缓冲区内保存一个特殊字符-1，表示文件结束。

第17行将readPos重置为-1，从文件中将数据加载到缓冲区后，从缓冲区的开始处读字符。

第19~20行累加读取位置readPos，并将缓冲区内readPos位置的字符保存到ch中。

第21~24行判断，若上一个字符是换行符，则将行号加1，列号清0。

第25~28行判断，若当前字符是文件结束符，则将文件关闭，文件指针置为NULL。之后，如果再次调用scan，根据第9行的判断，扫描器将仍返回特殊字符-1。

第29~30行判断，若当前字符不是换行符则将列号加1。

第31行将扫描过的字符保存到lastch，这样扫描下一个字符时，lastch总是保存了上次扫描到的字符。

第32行返回当前扫描到的字符。

通过对扫描器算法scan的不断调用，可以将源文件转化为线性的字符序列，为解析器提供输入。不过在展开对解析器的讨论前，需要明确词法记号的定义。

3.1.2 词法记号

词法记号是高级语言代码的基本单位，可以认为高级语言代码是词法记号按照一定规则的组合。词法记号通常可以分为标识符、关键字、常量、界符四大类，高级语言的定义对词法记号的定义有直接影响。不同语言对标识符的定义不同，如**Visual Basic**不区分标识符的大小写，**C**语言区分标识符的大小写。不同语言的关键字表也不尽相同，如在**C**语言内不存在**C++**的**virtual**关键字。不同语言的界符定义不同，**PASCAL**的赋值运算符为“:=”，而**C**语言的赋值运算符为“=”等。

本书编译系统处理的自定义语言的词法记号如下：

- 1) 类型系统。支持**int**、**char**、**void**基本类型和一维指针、一维数组类型。涉及的词法记号有关键字**int**、**char**、**void**，指针运算符为‘*’，取址运算符为‘&’，数组索引运算符为‘[’和‘]’。
- 2) 常量。字符常量、字符串常量、二/八/十进制整数。涉及的词法记号有数字常量**num**、字符常量**ch**、字符串常量**str**。与常量对应的变量使用标识符表示，因此标识符**id**也是词法记号。
- 3) 表达式。支持加、减、乘、除、取模、取负、自加、自减算术运算，大于、大于等于、小于、小于等于、等于、不等于关系运算和“与”、“或”、“非”逻辑运算。涉及的词法记号有‘+’，‘-’，‘*’

‘ , ’ , ‘ / ’ , ‘ % ’ , ‘ - ’ , ‘ ++ ’ , ‘ - ’ , ‘ > ’ , ‘ >= ’ , ‘ < ’ , ‘ <= ’ , ‘ == ’ , ‘ != ’ , ‘ && ’ , ‘ || ’ , ‘ ! ’ 。注意这里的乘法运算符和指针运算符是同一个字符，在词法分析器内它们被视为同一个词法记号。同理，减法运算符和取负运算符也是如此。

4) 语句。支持赋值语句，**do-while**、**while**、**for**循环语句，**if-else**、**switch-case**条件分支语句，函数调用、**return**、**break**、**continue**语句。涉及的词法记号有赋值运算符‘ = ’，关键字**do**、**while**、**for**、**if**、**else**、**switch**、**case**、**default**、**return**、**break**、**continue**。另外，复合语句或函数体需要使用花括号包含起来；基本语句都是以分号结束；**case**和**default**关键字后使用冒号分隔，因此词法记号还包含‘ { ’、‘ } ’、分号以及冒号。

5) 声明与定义。支持**extern**变量声明、函数声明，变量、函数定义。涉及的词法记号有**extern**关键字，‘ (’和‘) ’，注意小括号也可能出现在表达式中。另外，变量定义列表和函数的参数列表都使用逗号分隔，因此逗号是词法记号。

6) 其他。支持默认类型转换、单行和多行注释等。默认类型的转换属于代码生成部分的内容，注释不是有效的词法记号。不过除了以上提到的词法记号之外，还需要引入两个特殊的词法记号。**err**表示词法分析出错时返回的词法记号，词法分析器或语法分析器都自动忽略这个词法记号。此外，使用词法记号**end**表示文件结束。

于是，自定义语言涉及的所有词法记号如表3-1所示。

表3-1 词法记号

类别	词法记号	含义	类别	词法记号	含义
特殊	err	错误记号	界符	mul	*
	end	文件结束		div	/
标识符	id	标识符		mod	%
常量	num	数字		inc	++
	ch	字符		dec	--
	str	字符串		not	!
关键字	kw_int	int		lea	&
	kw_char	char		and	&&
	kw_void	void		or	
	kw_extern	extern		assign	=
	kw_if	if		gt	>
	kw_else	else		ge	>=
	kw_switch	switch		lt	<
	kw_case	case		le	<=
	kw_default	default		equ	==
	kw_while	while		nequ	!=
	kw_do	do		comma	,
	kw_for	for		colon	:
	kw_break	break		semicon	;
	kw_continue	continue		lparen	(
	kw_return	return		rparen)
界符	add	+		lbrac	{
	sub	-		rbrac	}

我们使用一个枚举类型记录所有的词法记号标签，为后面的代码提供符号定义。

```
1  /*
2   词法记号标签

3  */
4  enum Tag
```

```

{
5      ERR,                                //错误, 异常

6      END,                                //文件结束标记

7      ID,                                //标识符

8      KW_INT, KW_CHAR, KW_VOID,          //数据类型

9      KW_EXTERN,                          //extern
10     NUM, CH, STR,                      //常量

11     NOT, LEA,                          //单目运算
! & - *
12     ADD, SUB, MUL, DIV, MOD,          //算术运算符

13     INC, DEC,                          //自加自减

14     GT, GE, LT, LE, EQU, NEQU,        //比较运算符

15     AND, OR,                          //逻辑运算

16     LPAREN, RPAREN,                   //( )
17     LBRACK, RBRACK,                   //[ ]
18     LBRACE, RBRACE,                   //{ }
19     COMMA, COLON, SEMICON,            //逗号
, 冒号
, 分号

20     ASSIGN,                          //赋值

21     KW_IF, KW_ELSE,                   //if-else
22     KW_SWITCH, KW_CASE, KW_DEFAULT,   //swicth-case-
deault
23     KW_WHILE, KW_DO, KW_FOR,          //循环

24     KW_BREAK, KW_CONTINUE, KW_RETURN
//break, continue, return
25 };

```

注意Tag枚举类型内保存的词法记号标签都是大写，以避免与C语言关键字冲突。关键字标签都是以KW_开始，界符标签被分配了合适的名字。

词法记号标签只是区分了不同的词法记号，而对一些特殊的词法记号，如标识符、常量等除了需要保存词法记号标签信息外，还要保存标识符名称、常量值等信息，以用来构造符号表。我们采用面向对象的思想来设计与词法记号相关的类。

如图3-4所示，基类Token表示一般的词法记号，它只包含一个公有字段tag，用于记录词法记号的标签。Token有四个派生类Id、Num、Char、Str，分别对应标识符、数字常量、字符常量、字符串常量。其中Id的公有字段name记录了标识符的名称，Num的公有字段val记录了数字常量的值，Char的公有字段ch记录了字符常量的值，Str的公有字段str记录了字符串常量的值。

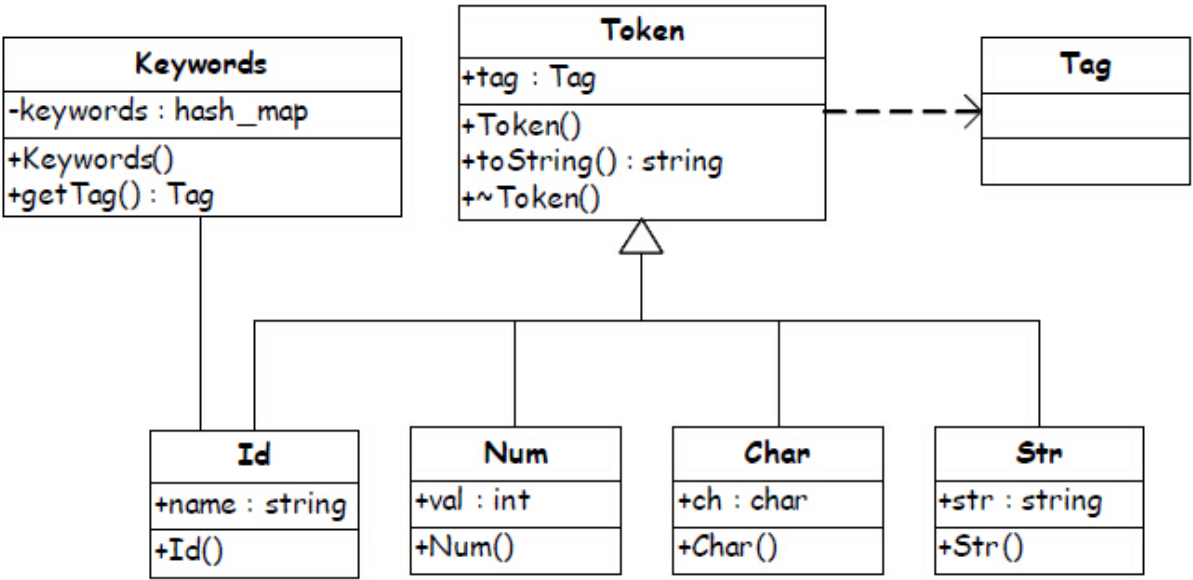


图3-4 词法记号类图

与Id关联的是Keywords类，它的公有字段keywords是教列表类型，记录了自定义语言定义的所有关键字。Keywords的实例化类型为hash_map<string, Tag, string_hash>，它记录了关键字名称与关键字词法记号标签的映射关系。如此设计的原因是关键字在词法分析器内可以看作一类特殊的标识符，词法分析器在创建标识符词法记号对象之前只需要使用getTag方法查询Keywords内保存的关键字表，便可以确定是创建Id对象还是普通的关键字词法记号Token对象。

按照上述类的设计，我们实现了词法记号相关的类，代码如下：

```
1  /*
2   词法记号类

3  */
4  class Token

5  {
6  public:
7      Tag tag;                                //内部标签

8      Token (Tag t);
9      virtual string toString();
10     virtual ~Token ();
11 };
12 /*
13  标识符记号类

14 */
15 class Id
:public Token
16 {
17 public:
18     string name;
19     Id (string n);
20     virtual string toString();
21 };
22 /*
23  数字记号类

24 */
25 class Num
```



```

:public Token
26 {
27 public:
28     int val;
29     Num (int v);
30     virtual string toString();
31 };
32 /*
33  字符记号类

34 */
35 class Char

:public Token
36 {
37 public:
38     char ch;
39     Char (char c);
40     virtual string toString();
41 };
42 /*
43  字符串记号类

44 */
45 class Str

:public Token
46 {
47 public:
48     string str;
49     Str (string s);
50     virtual string toString();
51 };
52
53 /*
54  关键字表类

55 */
56 class Keywords

57 {
58                                     //hash函数

59                                     //struct
string_hash{
60     size_t operator()(const string& str) const{
61         return __stl_hash_string(str.c_str());
62     }
63 };
64     hash_map<string, Tag, string_hash> keywords;
65 public:
66     Keywords();                                     //关键字表初始化

67     Tag getTag(string name);                       //测试是否是关键字

68 };

```

确定了词法记号后，接下来便是将扫描器输出的字符序列转化为词法记号的序列，这一步由解析器完成。如图3-1所示，解析器读入字符，使用有限自动机对输入字符进行匹配，最终输出词法记号。因此在描述解析器实现之前，还需要了解如何构造词法记号的有限自动机。

3.1.3 有限自动机

在编译原理的教材中，对有限自动机的描述十分详细，因此我们假定读者了解这方面的知识。有限自动机识别的语言称为正则语言，有限自动机分为确定的有限自动机（**DFA**）和非确定的有限自动机（**NFA**）两种。**DFA**和**NFA**都可以描述正则语言，**DFA**规定只能有一个开始符号，且转移标记不能为空，其代码实现较为方便。由于每个**NFA**都可以转化为一个**DFA**，本书的词法分析器统一使用**DFA**描述前面定义的所有词法记号。

1.标识符

在第2章中曾描述过标识符的有限自动机，作为词法分析过程的例子。对于任意**DFA**总有一个唯一的开始状态0，它的输入是一串字符序列，根据字符选择不同的转移进入下一个状态，当遇到结束状态时接收已输入的字符串。如图3-5所示，标识符从开始状态起，当读入下划线或字母时进入状态id，状态id是结束状态，其本身也可以接收任意多个下划线、字母和数字，当读入其他不能识别的字符时便停止自动机的识别过程。这正好符合C语言对标识符的定义：以下划线或字母开始的任意下划线、字母和数字组合的字符串。

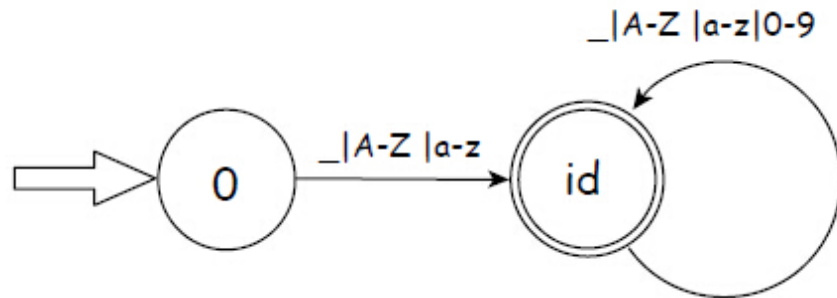


图3-5 标识符有限自动机

2.关键字

关键字是一类特殊的词法记号，本质上与标识符没有任何区别，只是词法分析器将之作为系统保留的标识符，不允许用户重新定义。我们在分析标识符结束后可以查询关键字表，来确定当前识别的标识符是普通的标识符还是关键字。表3-2描述了自定义语言中所有的关键字。

表3-2 关键字表

关键字	字符串值	词法标签
int	"int"	KW_INT
char	"char"	KW_CHAR
void	"void"	KW_VOID
extern	"extern"	KE_EXTERN
if	"if"	KW_IF
else	"else"	KW_ELSE
switch	"switch"	KW_SWITCH
case	"case"	KW_CASE
default	"default"	KW_DEFAULT
while	"while"	KW_WHILE
do	"do"	KW_DO
for	"for"	KW_FOR
break	"break"	KW_BREAK
continue	"continue"	KW_CONTINUE
return	"return"	KW_RETURN

3.常量

常量词法记号有三种：`num`、`ch`和`str`，它们分别对应数字常量、字符常量和字符串常量。以下分别描述这三种常量的自动机结构。

对于数字常量，本书仅考虑整数常量，支持十进制、二进制、八进制和十六进制整数。整数的定义可以简单地认为是数字0~9的任意组合。对不同进制的整数的定义如下。

1) 十进制整数：以数字1~9开始，0~9中任意个数字组合的字符串。

2) 八进制整数：以数字0开始，0~7中任意个数字组合的字符串。这也是为什么十进制整数不能以0开始，因为会与八进制整数的定义冲突。

突。

3) 二进制整数：以字符串“0b”开始，0~1中任意个数字组合的字符串。

4) 十六进制整数：以字符串“0x”开始的，0~9及字母a~f、A~F中一个或多个数字、字母组合的字符串。

整数的有限自动机如图3-6所示，其中结束状态d-num、o-num、b-num和h-num分别表示十进制、八进制、二进制、十六进制整数的自动机结束状态。结束状态err表示自动机识别过程中出现词法错误时到达的状态，一旦进入err状态便停止自动机，报告对应的词法错误。整数有限自动机从状态0开始，当读入字符1~9时，进入d-num状态进行十进制整数的识别。当读入字符0时转移到o-num状态，然后继续读入字符，如果是0~7则识别八进制整数，如果读入字符b则进行二进制整数的识别，如果读入字符x则进行十六进制整数的识别。

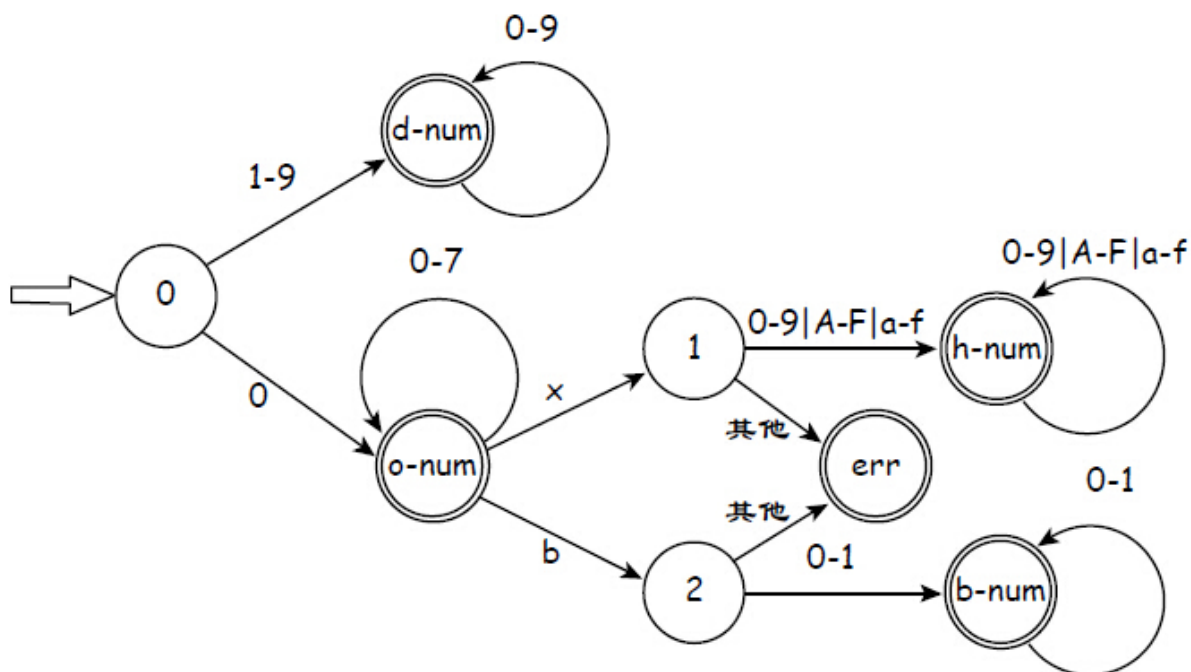


图3-6 整数有限自动机

字符常量是用左右单引号包含的一个字符，并且支持特殊字符的转义。比如‘a’和‘\n’都是合法的字符。

图3-7描述了字符有限自动机的结构，其中状态ch为识别字符的结束状态，err为错误状态。字符有限自动机从状态0开始，读入一个单引号字符进入状态1。再次读入一个普通字符进入状态3，或者读入字符‘\’和另一个字符进行转义字符的识别，如果读入的字符是换行符、文件结束符或单引号则报错。我们定义的字符转义字符包括‘\n’、‘\\’、‘\t’、‘\0’和‘\’，换行符和文件结束符是不能转义的，未定义的转义字符作为普通字符对待，转义字符的处理在状态2处完成。

状态3表示识别了一个正常的字符，然后再读入一个单引号完成字符词法记号的识别，除此之外则报告词法错误。

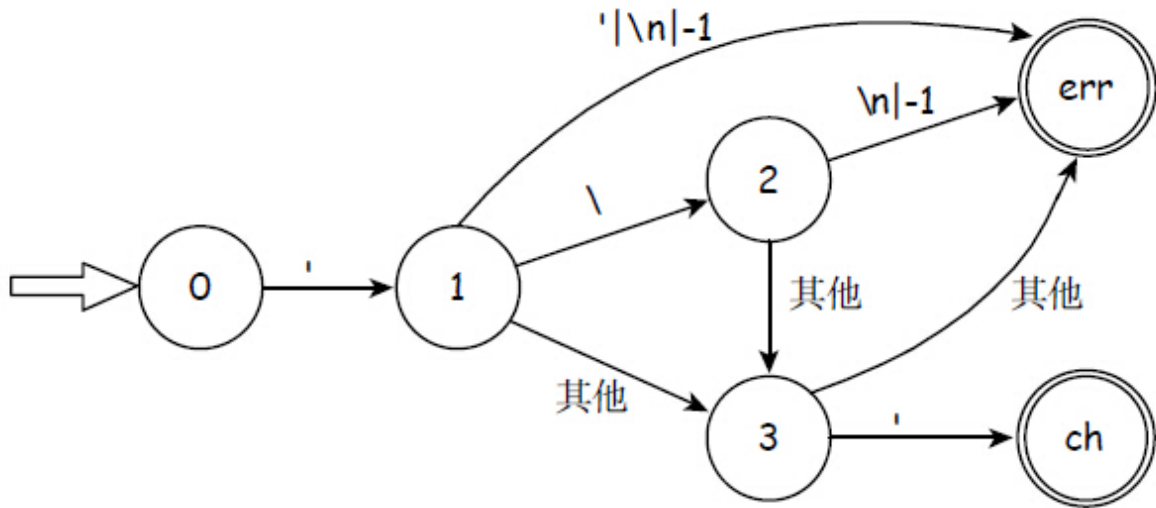


图3-7 字符有限自动机

字符串有限自动机如图3-8所示，其中状态str为识别字符串的结束状态，err为错误状态。字符串有限自动机从状态0开始，读入一个双引号字符进入状态1。状态1可以接收任意多个普通字符，如果此时读入字符'\'和另一个字符，则进入状态2进行转义字符的识别，如果读入的字符是换行符、文件结束符则报错。我们定义的字符串转义字符包括'\n'、'\'、'\t'、'\0'、'\换行'和'\\"'，其中'\换行'是对换行符转义，表示字符串内换行，文件结束符是不能转义的，未定义的转义字符作为普通字符对待，转义字符的处理在状态2处完成后回到状态1继续处理。处于状态1时，只要读入一个双引号便完成了字符串词法记号的识别。

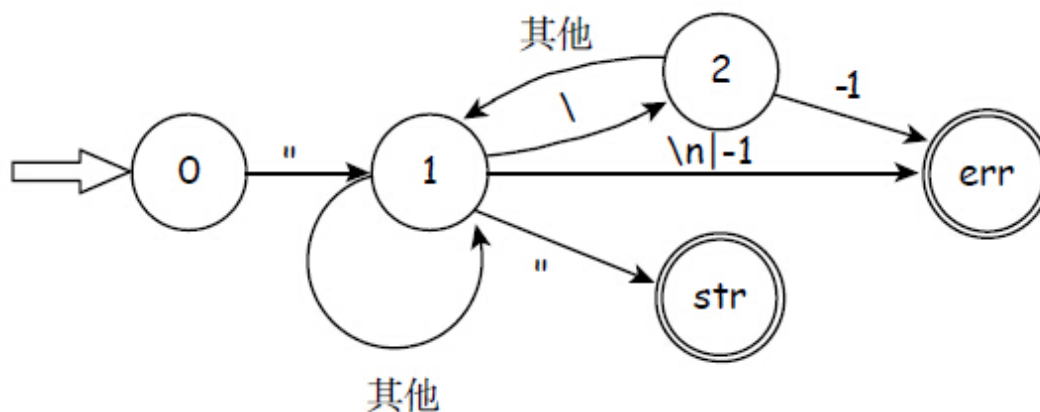


图3-8 字符串有限自动机

4.界符

词法记号中的界符数量较多，但是形式无外乎两种：单字节界符和双字节界符。比如“%”是单字节界符，“>=”是双字节界符。界符的有限自动机结构比较简单。

单字节界符有限自动机如图3-9a所示，取模运算符有限自动机读入一个字符‘%’后进入结束状态，完成词法记号mod的识别。双字节界符有限自动机如图3-9b所示，大于/大于等于运算符有限自动机读入字符‘>’进入结束状态gt，此时产生词法记号gt。但是按照“贪心”的原则，此时自动机如果读入字符‘=’，则进入结束状态ge，产生词法记号ge。其他类型的界符的有限自动机结构类似，这里就不必一一列举了。

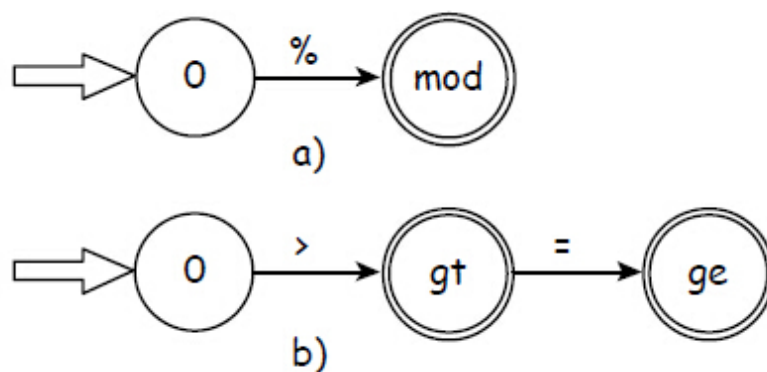


图3-9 界符有限自动机

我们设计的词法记号中单字节界符包括‘+’、‘-’、‘*’、‘/’、‘%’、‘&’、‘>’、‘<’、‘=’、‘!’、‘,’、‘:’、‘;’、‘(’、‘)’、‘[’、‘]’、‘{’、‘}’和文件结束符-1，双字节界符包括‘++’、‘--’、‘>=’、‘<=’、‘==’、‘!=’、‘&&’和‘||’。需要注意的是，界符‘/’除了作为除法运算符外，还可以作为单行/多行注释的开始。界符‘||’在读入第一个字符‘|’时并不能被自动机接收，因为我们没有定义词法记号‘|’。

5.无效词法记号

除了以上的词法记号外，还有两类自动机没有涉及，因为它们并不产生真正的词法记号，我们称为无效词法记号。一类是空白字符

（空格、制表符、换行符），另一类是注释。参考C语言的特点，所有有效词法记号之间可以出现任意多个空白字符和注释。

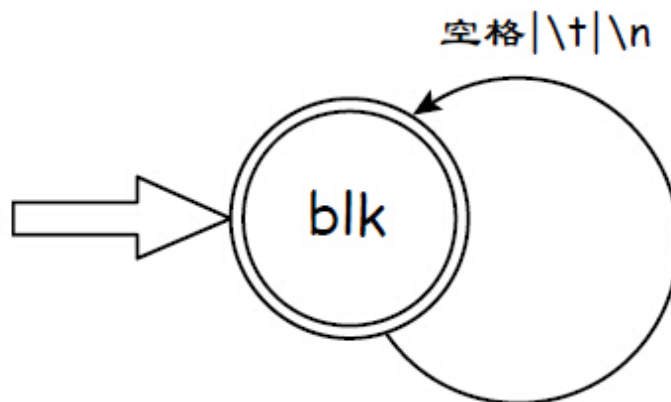


图3-10 空白字符有限自动机

如图3-10，空白字符的有限自动机非常简单，它只有一个状态，即开始状态亦是结束状态，并接收任意多个空格、‘\t’和‘\n’。前面描述的有限自动机识别结束后都会产生一个词法记号，那么空白字符的有限自动机识别结束后应该如何处理呢？有两种方式可以选择，一种是不产生任何词法记号，识别结束后继续识别其他词法记号。另一种方式是产生词法记号err，因为词法记号err会被词法分析器或语法分析器自动忽略。前面的有限自动机在识别出错时都会产生词法记号err，因此不会影响后续词法记号的识别。

如图3-11所示，注释有限自动机从开始状态0读入字符‘/’后进入结束状态div，此时可以识别除法运算词法记号div。但是按照“贪心”规则，如果此时读入字符‘/’或‘*’则进入单行/多行注释的识别过程。单行注释内容在状态1处处理，此时读入任意一行内容，直到遇到换行符或文件结束符后进入结束状态s-com，识别单行注释。多行注释内容在状态2处处理，此时可以读入任意长度的内容，当读入文件结束

符时产生词法错误，当读入字符‘*’时进入状态3。状态3处如果读入字符‘/’进入结束状态m-com，识别多行注释，如果仍读入字符‘*’则继续保持在状态3，否则返回状态2。状态3可以接收任意多个字符‘*’是为了避免出现“/*.....**/”字符串不能被识别为注释的情况。这里需要注意的是，多行注释不能出现嵌套的情况，嵌套多行注释不能被有限自动机识别，因为超出了正则文法的描述能力，属于上下文无关文法。

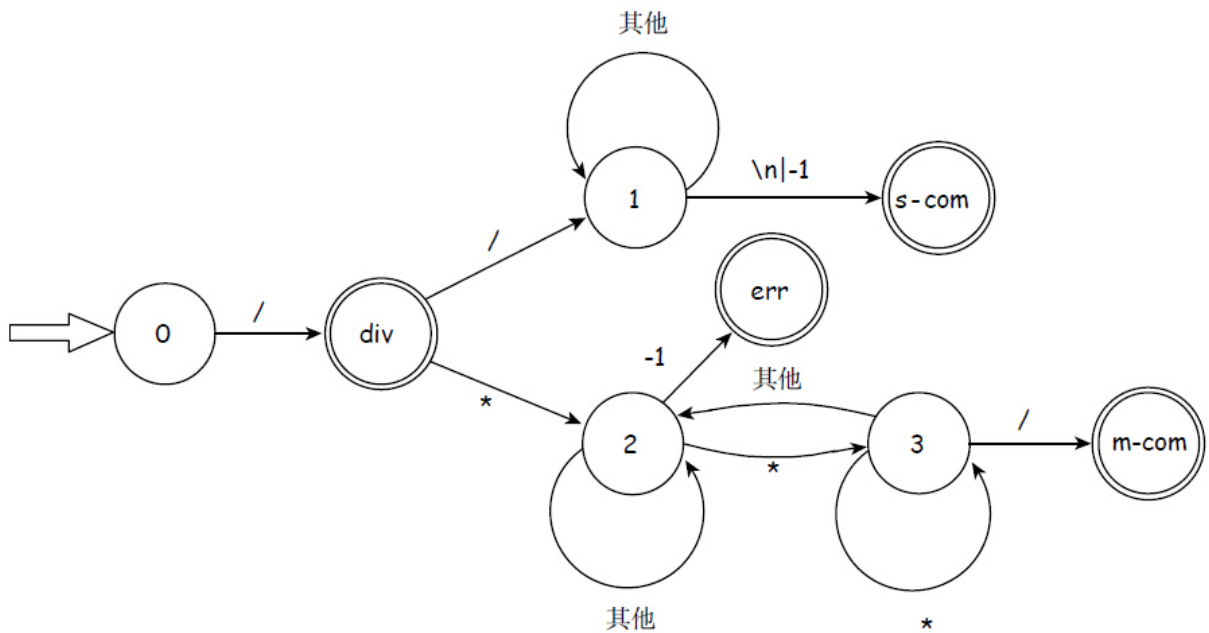


图3-11 注释有限自动机

无论是单行注释还是多行注释，识别结束后都会返回词法记号 err。这与空白字符有限自动机的处理有所区别，因为注释的有限自动机包含了除法运算符词法记号的识别，为了保持代码的一致性，该自动机不得不返回一个词法记号。

3.1.4 解析器

解析器的输入是扫描器产生的线性字符序列，而输出是词法记号序列。解析器的构造依赖于词法记号有限自动机的实现，词法记号的有限自动机构造完毕后，便可以编码实现解析器（真正意义上的词法分析器），完成词法记号的解析。

根据有限自动机实现词法分析器一般有两种方式：基于表驱动的方式和硬编码方式。基于表驱动方式的词法分析需要为词法记号建立状态转移表，而硬编码方式的词法分析则使用程序控制结构直接实现词法分析。

1.基于表驱动的词法分析

表3-3是有限自动机的状态转移表，这里主要描述了标识符有限自动机的状态转移，其他词法记号有限自动机的状态和转移被简化处理。表格的第一列表示自动机的当前状态，表格的第一行表示自动机读入的当前字符，即状态转移弧上的字符，单元格内表示自动机将要进入的下一个状态，这个关系正好与自动机的基本结构对应。

表3-3 状态转移表

转移 状态 \ 字符	-	a-z	A-Z	0-9	其他
0	id	id	id	数字	其他
id	id	id	id	id	accept

自动机从状态0开始，如果读入的字符是‘_’或‘a-z’中的任意一个小写字母，或‘A-Z’中的任意一个大写字母，查询状态转移表得到下一个状态为id。转移到状态id，继续读入的字符如果是‘_’或小写字母、大写字母或‘0-9’中的数字，则仍进入状态id，否则转移到accept状态。accept是一个特殊的状态，它表示除了当前读入字符之外的所有已读入字符构成自动机识别的字符串。此时词法分析器根据进入accept状态的那个状态（状态id）决定词法记号的处理结果，即产生标识符的词法记号。

需要注意的是，在进入accept状态时读入的字符并不是自动机已经接受的字符。因此，在后续的词法记号自动机识别的过程中，需要重新读入当前字符，以避免当前读入字符被跳过。为此，每次查询状态转移表之前并不读入新的字符，而是假定字符已经读入。那么自动机开始运行时，需要将当前字符初始化为空格（或者‘\n’，‘\t’），这样自动机启动后会首先进入空白字符有限自动机的处理，识别这个空白字符。

基于表驱动的词法分析的伪代码描述如下：

```

1  cur_char=' ';
   字符为空格
//初始

```

```

2 Token* Lexer::tokenize
   {}                                     //词法记号解析

3         cur_state=0;                                     //初始
   状态为

0
4         while(1){                                     //启动
   有限自动机

5                 next_state=table[cur_state,cur_char];   //查表
   获取下一个状态

6                 if(next_state==accept){                 //接受
   状态

7                 return process
   (cur_state);                                     //处理接受状态

8                 }
9                 else if(next_state==error){             //错误
   状态

10                return lex_error
   (cur_state,cur_char);                         //词法错误处理

11                }
12                else{                                     //正常
   状态转移

13                handle_state
   (cur_state,cur_char);                         //处理当前状态

14                cur_state=next_state;                   //进入
   下一个状态

15                cur_char=scan
   (file);                                       //扫描获取下一个字符

16                }
17        }
18 }

```

第1行将当前读入字符初始化为空格，这样第一次调用`tokenize`时会执行空白字符自动机识别的过程。

第3行将当前状态初始化为开始状态0，开始词法记号的解析过程。

第5行查询状态转移表`table`，行索引为`cur_state`，列索引为`cur_char`。

第6~8行处理`accept`状态，返回词法记号。

第9~11行处理`err`状态，进行词法错误处理。

第12~16行处理正常的状态转移，首先处理当前状态和已经接受的字符，比如计算数字常量的值、处理转义字符等。然后设定`cur_state`为查询得到的下一个状态，调用扫描器读入下一个字符，回到第6行继续词法记号解析。

使用状态转移表进行词法分析的实现就是查询状态转移表、状态比较和状态处理的过程，实现简单。词法分析器的自动生成工具一般都采用这种方法。词法分析器自动生成工具根据用户提供的词法记号定义配置文件，建立词法记号有限自动机NFA，然后将NFA确定化为DFA，再将DFA最小化，生成状态转移表，最后按照上述过程进行词法

分析。然而，主流的编译器并未采用表驱动的词法分析方式，而是采用硬编码的方式，可能考虑了以下因素：

1) 保存状态转移表需要大量的存储空间，构建状态转移表对词法记号的定义有很强的依赖，一旦更改了词法记号的定义，状态转移表变化很大，不利于代码维护。

2) 基于表驱动的词法分析过程形式虽然简单，但是灵活性较差，不利于对特定状态的自定义处理。所有词法记号有限自动机的状态转移在代码层面形式基本相同，导致代码的可读性较差，调试不够方便。

3) 基于表驱动的词法分析在每次读入一个字符后都会发生状态转移，大量的查表、状态比较和状态处理降低了词法分析器的性能。

因此，本书采用硬编码的方式构建词法分析器。

2.硬编码方式的词法分析

与基于表驱动方式的词法分析不同的是，硬编码方式的词法分析不需要显式地确立有限自动机的状态，它使用程序的控制结构直接对词法记号进行解析，即根据词法记号本身的含义，使用代码解析词法记号的内容。

我们仍以标识符为例，再次分析标识符的定义：以下划线、字母开始的任意多个下划线、字母和数字组合的字符串。首先对一个标识符从概念上进行拆分，一个合法的标识符包含两部分：标识符的第一部分是一个字符，它可能是下划线或字母。标识符的第二部分可以是任意长度的字符串，但是字符串内的每个字符必须是下划线、字母或数字。根据这样的拆分，很容易发现解析标识符的控制结构。首先是一个判断语句，判定读入的字符是不是下划线或字母，以确定开始识别标识符。然后是一个循环语句，期望读入更多的下划线、字母或数字。其伪代码描述如下：

```
if(ch==下划线
or字母
){
    ch=scan(file);
    while(ch==下划线
or字母
or数字
    ){
        ch=scan(file);
    }
}
```

这样的硬编码方式与表驱动方式识别出的字符串完全等价，而且不需要考虑有限自动机的状态转移，也完全消除了查表、判断状态等操作。其实，硬编码中的程序控制语句已经蕴含了有限自动机状态的转移信息。

我们对有限自动机做如下处理：将有限自动机的状态转化为一个标号，将每条状态转移弧转化为一条跳转到目标状态的goto语句，且在每条goto语句前读入新的字符，弧上的字符标签转化为判断条件。对于标识符的有限自动机，我们可以得到如下代码：

```
0:      if(ch==下划线
or字母
){
        ch=scan(file);
        goto 1;
      }
      goto end;
1:      if(ch==下划线
or字母
or数字
){
        ch=scan(file);
        goto 1;
      }
      goto end;
end:
```

然后将这段代码转化为控制流图，如图3-12所示。

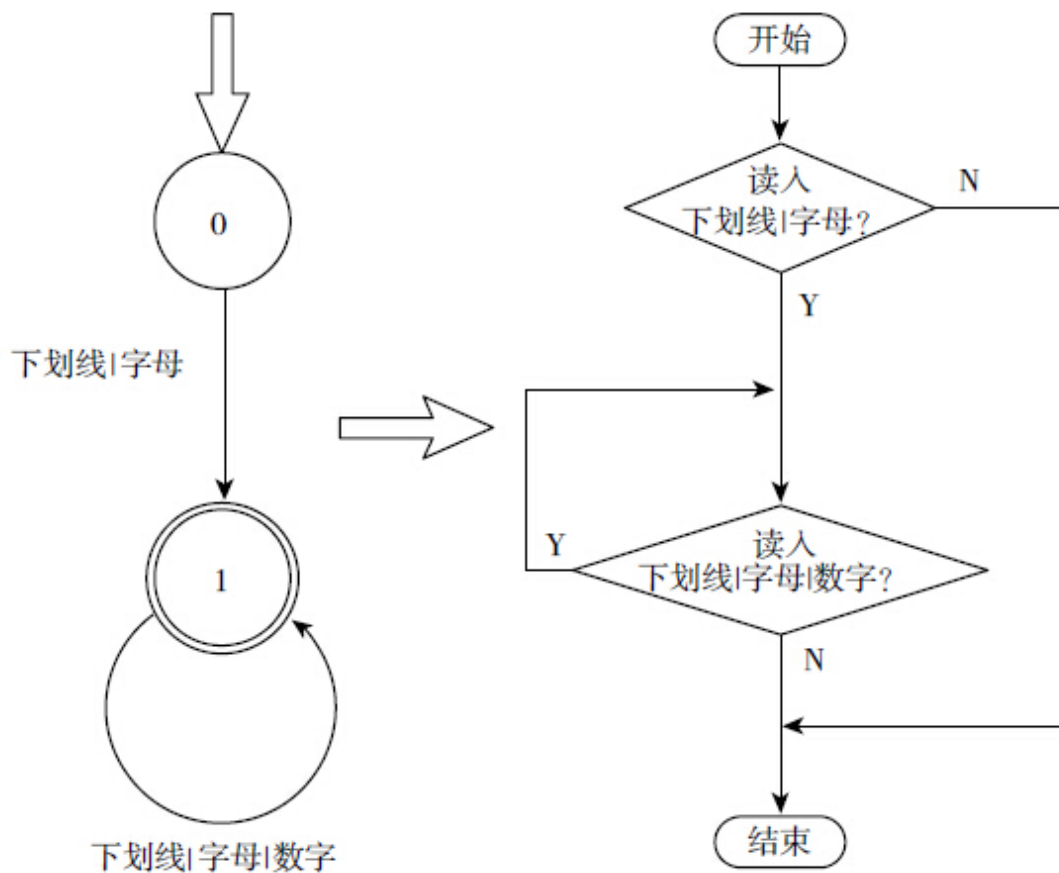


图3-12 DFA与控制流图

我们发现，根据DFA转化得到的代码控制流图与前面硬编码的程序控制结构完全一致，这种将DFA直接转化为编码的方式一般称为直接编码方式。虽然使用这种方式也可以完成词法分析器的构造，但是这种方式仍需要考虑自动机状态的转换，并且包含大量的goto语句，代码明显不如硬编码方式简洁。因此，使用硬编码方式的词法分析是较好的选择。接下来逐一描述词法记号的硬编码实现。

(1) 标识符

在前面的章节中已经描述了标识符词法记号的硬编码实现。然而，在实际的词法分析过程中，硬编码的程序控制结构只是提供了识别词法记号的框架。对于标识符来说，在词法分析过程中，还需要记录标识符的名称，创建标识符词法记号对象，这需要在硬编码控制结构中插入相关代码来完成。识别标识符词法记号的代码如下：

```
1  if(ch>='a'&&ch<='z' || ch>='A'&&ch<='Z' || ch=='_'){
2      string name="";
3      do{
4          name.push_back(ch);
5          //记录字符

6          scan();
7          //读入字符

8      }while(ch>='a'&&ch<='z' || ch>='A'&&ch<='Z'
9          || ch=='_' || ch>='0'&&ch<='9');
10         //匹配结束

11     Tag tag=keywords.getTag(name);
12     //查询关键字

13     if(tag==ID)
14         //正常的标识符

15         t=new Id(name);
16         //创建标识符

17     else
18         //关键字

19         t=new Token(tag);
20         //创建关键字

21 }
```

标识符词法分析过程中，使用**name**记录接收的字符。这里仍假定当前字符已经提前读入，存储在**ch**内。需要注意的是，前面描述的标识符识别的程序控制结构是**if+while**形式，而这里是**if+do-while**形式。

因为无论while语句条件是否成立，都会执行第4~5行语句，因此这里可以修改为do-while循环。这也从侧面说明了硬编码实现的词法分析器的编码灵活性。

代码的第1~7行完成了标识符词法记号的识别，并将标识符的名字记录在变量name中。第8行通过查询关键字表来确定当前识别的字符串是普通的标识符还是系统保留的关键字。

(2) 关键字

我们一直把关键字当作一类特殊的标识符对待，甚至前面识别标识符的代码中都包含了关键字词法记号的生成。在词法分析器的实现中，使用了散列表保存关键字信息，相关代码如下：

```
1  /*
2   关键字列表初始化

3  */
4  Keywords::Keywords()
5  {
6      keywords["int"]=KW_INT;
7      keywords["char"]=KW_CHAR;
8      keywords["void"]=KW_VOID;
9      keywords["extern"]=KW_EXTERN;
10     keywords["if"]=KW_IF;
11     keywords["else"]=KW_ELSE;
12     keywords["switch"]=KW_SWITCH;
13     keywords["case"]=KW_CASE;
14     keywords["default"]=KW_DEFAULT;
15     keywords["while"]=KW_WHILE;
16     keywords["do"]=KW_DO;
17     keywords["for"]=KW_FOR;
18     keywords["break"]=KW_BREAK;
19     keywords["continue"]=KW_CONTINUE;
20     keywords["return"]=KW_RETURN;
21 }
22 /*
23  测试是否是关键字

24 */
```

```
25 Tag Keywords::getTag
   (string name)
26 {
27     return keywords.find(name)!=keywords.end()
28         ?keywords[name]:ID;
29 }
```

在Keywords类的构造函数中，我们初始化了关键字的散列表keywords。当词法分析器需要确定一个字符串是否是关键字时，只需要调用Keywords类的getTag方法即可。第27~28行是一个条件表达式，它在关键字表内查询name是否存在，如果存在则返回散列表记录的关键字标号，否则返回标识符标号ID。只要在该函数的调用处判断函数getTag返回的词法标签是否是ID，便能确定当前识别的字符串是标识符还是关键字。

(3) 常量

我们仍按照数字常量、字符常量、字符串常量的顺序介绍常量词法记号的识别。与标识符类似，仍然是对词法记号在概念上进行拆分，以确定识别词法记号的程序控制结构。

数字常量支持四种进制：十进制、八进制、二进制和十六进制，只要以数字0~9开始的字符串都会产生数字词法记号。十进制整数要求以1~9开始，是任意多个0~9数字的组合。这与标识符类似，是一个if+do-while控制结构。八进制整数要求以“0”开始，二进制整数要求以“0b”开始，十六进制整数要求以“0x”开始。它们拥有公共的前缀0，因此还需要读入一个字符来确定具体的数字进制。如果读入字符‘b

’，则确定是二进制整数，后边紧跟以0~1开始的任意0~1组合的字符串，也是一个if+do-while控制结构。如果读入字符’x’，则确定是十六进制整数，后边紧跟以0~9、A~Z或a~z开始的，任意0~9、A~Z或a~z组合的字符串，也是if+do-while控制结构。如果读入的是0~7中的字符，则确定是八进制整数，后边紧跟任意多个0~7数字的组合，是一个do-while控制结构。如果读入的是其他字符，则表示仅有一个数字0被接受。实现代码如下：

```
1  if(ch>='0'&&ch<='9'){
2      int val=0;
3      if(ch!='0'){
4          //十进制
5          do{
6              val=val*10+ch-'0';
7              scan();
8          }while(ch>='0'&&ch<='9');
9      }
10     else{
11         scan();
12         if(ch=='x'){
13             //十六进制
14             scan();
15             if(ch>='0'&&ch<='9' || ch>='A'&&ch<='F'
16                 || ch>='a'&&ch<='f'){
17                 do{
18                     val=val*16+ch;
19                     if(ch>='0'&&ch<='9')val-=10;
20                     else if(ch>='A'&&ch<='F')val+=10-'A';
21                     else if(ch>='a'&&ch<='f')val+=10-'a';
22                     scan();
23                 }while(ch>='0'&&ch<='9' || ch>='A'&&ch<='F'
24                     || ch>='a'&&ch<='f');
25             }
26             else{
27                 LEXERROR(NUM_HEX_TYPE);
28             }
29             //0x后无数据
30         }
31     }
32     t=new Token(ERR);
33 }
34 else if(ch=='b'){
35     //二进制
36     scan();
37 }
```



```

31             if(ch>='0'&&ch<='1'){
32                 do{
33                     val=val*2+ch-'0';
34                     scan();
35                 }while(ch>='0'&&ch<='1');
36             }
37             else{
38                 LEXERROR(NUM_BIN_TYPE);
39             }
40         }
41     }
42     else if(ch>='0'&&ch<='7'){ //八
43         do{
44             val=val*8+ch-'0';
45             scan();
46         }while(ch>='0'&&ch<='7');
47     }
48 }
49 if(!t)t=new Num(val); //最终数字

50 }

```

第2行使用变量`val`保存数字的值，初始化为0。

第3~8行是十进制整数的识别过程，第9~48行是其他进制数的识别过程。

第11~28行是十六进制整数的识别过程，第24~27行处理十六进制整数定义时，“0x”之后无有效字符的词法错误。LEXERROR宏用于输出词法错误信息，后面会对该宏进行介绍，并产生词法记号`err`。

第29~41行是二进制整数的识别过程，第37~40行处理二进制整数定义时，“0b”之后无有效字符的词法错误。LEXERROR宏用于输出词法错误信息，产生词法记号`err`。

第42~47行是八进制整数的识别过程，此处不会产生词法错误。这是因为在进行非十进制整数识别时，已经读入字符‘0’，无论新读入的字符是否是0~7数字，0本身就是一个合法的八进制整数。

第50行根据数字识别过程中计算得到的val的值创建数字词法记号。

对于字符常量，要求它是由两个单引号包含起来的唯一字符或一个转义字符。词法分析器读入一个单引号后开始对字符常量进行识别。如果读入字符‘\’则继续读入一个字符，并处理字符的转义。如果读入另一个单引号，则说明两个单引号之间没有有效字符，视为词法错误。如果读入了换行符或文件结束符，将导致词法错误。这个过程是一个典型的if控制结构。如果被转义的字符是换行符或文件结束符，则字符词法记号识别失败。否则进行正常的转义字符处理。这也是一个if控制结构，实现代码如下：

```
1  if(ch=='\'){
2      char c;
3      scan();
4      if(ch=='\\'){                                //转义

5          scan();
6          if(ch=='n')c='\n';
7          else if(ch=='\\')c='\\';
8          else if(ch=='t')c='\t';
9          else if(ch=='0')c='\0';
10         else if(ch=='\')c='\';
11         else if(ch==-1||ch=='\n'){                //文件结束

换行

12             LEXERROR(CHAR_NO_R_QUTION);
13             t=new Token(ERR);
14         }
15         else c=ch;                                //没有转义
```

```

16     }
17     else if(ch=='\n' || ch==-1){           //换行
//文件结束

18         LEXERROR(CHAR_NO_R_QUTION);
19         t=new Token(ERR);
20     }
21     else if(ch=='\'){                       //没有数据

22         LEXERROR(CHAR_NO_DATA);
23         t=new Token(ERR);
24         scan();                             //读掉引号

25     }
26     else c=ch;                             //正常字符

27     if(!t){
28         if(scan('\')){                     //匹配右侧引号
//读掉引号

29         t=new Char(c);
30         }
31         else{
32             LEXERROR(CHAR_NO_R_QUTION);
33             t=new Token(ERR);
34         }
35     }
36 }

```

第2行使用变量c记录识别的字符。

第4~16行识别转义字符，第11~14行处理不能转义的字符，报告词法错误。

第17~20行处理字符词法记号内出现换行符或文件结束符时的词法错误。

第21~25行处理两个单引号之间无有效字符的情况。注意这里识别右单引号后需要主动读入下一个字符，不再将这个右单引号作为下一

个词法记号的开始。

第27~35行识别字符词法记号的右单引号，第29行产生字符词法记号，第31~34行处理右单引号丢失的词法错误。

字符串常量与字符常量的识别相似，要求它是由两个双引号包含起来的字符和转义字符的任意组合，包括空串。词法分析器读入一个双引号后开始，并期望读入另一个双引号完成字符串常量的识别，这是一个while控制结构。如果读入字符'\'则继续读入一个字符，处理字符的转义。如果读入了换行符或文件结束符，将导致词法错误。这个过程是一个if控制结构。如果被转义的字符是文件结束符，则字符串词法记号识别失败。否则进行正常的转义字符处理。这也是一个if控制结构，实现代码如下：

```
1  if(ch=='"){
2      string str="";
3      while(!scan('"')){
4          if(ch=='\\'){                                //转义

5              scan();
6              if(ch=='n')str.push_back('\n');
7              else if(ch=='\\')str.push_back('\\');
8              else if(ch=='t')str.push_back('\t');
9              else if(ch=='"')str.push_back('"');
10             else if(ch=='\0')str.push_back('\0');
11             else if(ch=='\n');                        //字符串换行

12             else if(ch==-1){
13                 LEXERROR(STR_NO_R_QUTION);
14                 t=new Token(ERR);
15                 break;
16             }
17             else str.push_back(ch);
18         }
19         else if(ch=='\n' || ch==-1){                    //文件结束

20             LEXERROR(STR_NO_R_QUTION);
21             t=new Token(ERR);
```

```
22             break;
23         }
24         else
25             str.push_back(ch);
26     }
27     if(!t)t=new Str(str);           //最终字符串

28 }
```

第2行使用变量str记录识别的字符串。

第4~18行识别转义字符，第12~16行处理不能转义的字符，报告词法错误。

第19~23行处理字符串词法记号内出现换行符或文件结束符时的词法错误。

第27行产生字符串词法记号。

(4) 界符

我们按照界符词法记号的长度将界符分为两类：单字节界符和双字节界符。单字节界符的识别非常简单，直接根据读入的字符确定界符的词法标签，创建词法记号对象，然后读入下一个字符作为后继词法记号识别的开始。例如单字节界符‘%’，当读入字符‘%’后直接创建词法记号‘%’。使用伪代码描述如下：

```
if(ch=='%'){
    t=new Token(MOD);
    scan();
}
```

而对于双字节界符的识别，需要读入两个字符以确定界符的词法标签。例如双字节界符‘>=’，需要在读入字符‘>’后，再次读入一个字符，并判断是否是字符‘=’来确定识别的词法记号是‘>’还是‘>=’。这里需要注意的是，如果读入字符‘>’后，再次读入的字符不是‘=’，则产生‘>’词法记号，后续的词法记号从当前读入的字符开始继续识别。如果再次读入的字符是‘=’，则产生‘>=’词法记号，后续的词法记号应该从下一个字符开始继续识别，即词法分析器的“贪心”规则，通过调用扫描器获取下一个字符。使用伪代码描述‘>=’词法记号的识别过程如下：

```
if(ch=='>'){ //进入'
    >' 或'
    >=' 的识别

    Tag tag=GT; //暂时确定为'
    >'

    scan(); //读入下个字符

    if(ch=='='){ //判定是否是'
    >=

        tag=GE; //重新确定为'
    >=

        scan(); //读入下一个字符

    }
    t=new Token(tag); //创建词法记号

}
```

使用这样的方式识别双字节界符比较繁琐，为简化识别双字节界符的过程重新设计封装scan的接口，代码如下：

```
1  /*
2   封装的扫描方法

3  */
4  bool Lexer::scan
(char need=0)
5  {
6   ch=scanner.scan(); //扫描出字符

7   if(need){
8       if(ch!=need) //与预期不吻合

9       return false;
10      ch=scanner.scan(); //与预期吻合，扫描
下一个

11      return true;
12  }
13      return true;
14 }
```

重新封装的scan附加了参数need，并默认为0，这样直接调用scan()时与原本的scan功能相同。如果调用scan时指定了参数，则判断扫描出的字符是否与need相等。如果不等则返回false，表示扫描出的字符与指定参数need不匹配，否则继续扫描，读入新的字符，返回true。这样，无论读入的字符是否与参数匹配，当前字符位置都保存了下一个词法记号开始匹配的字符。使用重新封装的scan函数处理上述‘>=’词法记号的识别可以用条件表达式完成。

```
if(ch=='>'){ //进入
> 或
>= 的识别
```

```
t=new Token(scan('=')?GE:GT);
```

```
//自动处理词法记号的种类
```

```
}
```

将所有的界符的识别过程放在一个switch-case语句内，得到的实现代码如下：

```
1 switch(ch){//界符
```

```
2     case '+':
3         t=new Token(scan('+')?INC:ADD);break;
4     case '-':
5         t=new Token(scan('-')?DEC:SUB);break;
6     case '*':
7         t=new Token(MUL);scan();break;
8     case '/':
```

```
9         t=new Token(DIV);scan();break;
10    case '%':
11        t=new Token(MOD);scan();break;
12    case '>':
13        t=new Token(scan('=')?GE:GT);break;
14    case '<':
15        t=new Token(scan('=')?LE:LT);break;
16    case '=':
17        t=new Token(scan('=')?EQU:ASSIGN);break;
18    case '&':
19        t=new Token(scan('&')?AND:LEA);break;
20    case '|':
```

```
21        t=new Token(scan('|')?OR:ERR);
22        if(t->tag==ERR)
23            LEXERROR(OR_NO_PAIR);
```

```
//|| 没有一对
```

```
24        break;
25    case '!':
26        t=new Token(scan('=')?NEQU:NOT);break;
27    case ',':
28        t=new Token(COMMA);scan();break;
29    case ':':
30        t=new Token(COLON);scan();break;
31    case ';':
32        t=new Token(SEMICON);scan();break;
33    case '(':
34        t=new Token(LPAREN);scan();break;
35    case ')':
36        t=new Token(RPAREN);scan();break;
37    case '[':
38        t=new Token(LBRACK);scan();break;
39    case ']':
40        t=new Token(RBRACK);scan();break;
41    case '{':
42        t=new Token(LBRACE);scan();break;
```



```

43     case '}':
44         t=new Token(RBRACE);scan();break;
45     case -1:
46         t=new Token(END);scan();break;
47     default:

48         t=new Token(ERR);                                //错误的
词法记号

49         LEXERROR(TOKEN_NO_EXIST);
50         scan();
51 }

```

可见使用封装后的`scan`实现的界符解析代码更加简洁，不过仍有几点需要注意。

第8行处理`'/'`字符时，还未考虑注释的解析，稍后会详细介绍。

第20行处理`'|'`字符时，若不能匹配词法记号`'||'`，则产生词法错误，因为我们没有定义词法记号`'|'`。

第46行处理不符合文法定义的字符的情况，统一视作错误词法记号。

(5) 无效词法记号

每次进行词法记号识别之前，词法分析器会尽可能忽略空白字符，通过一个`while`循环结构很容易做到这点。

```

1  while(ch==' '||ch=='\n'||ch=='\t')                    //忽略空白符

2      scan();
3
词法记号
处理其他词法记号的识别

```

对于注释，可以从概念上拆分，描述它的识别过程。注释分为单行注释和多行注释，单行注释以‘//’引导，多行注释以‘/*’引导，它们包含公共的前缀‘/’，把它们与除法运算符词法记号‘/’放在一起处理。因此词法分析器读入字符‘/’后需要再读入一个字符，来确定是否是注释。

如果新读入的字符是‘/’，则确定为单行注释。此后词法分析器可以接收任意多个任意字符，直到遇到换行符或文件结束为止。使用while循环结构可以处理单行注释的识别。

如果新读入的字符是‘*’，则确定为多行注释。词法分析器可以接收任意多个非文件结束符字符，直到遇到字符‘*’时才进行后继的处理，这个过程使用while循环处理。在多行注释处理过程中，如果遇到字符‘*’，还需要尝试跳过连续的‘*’字符，这是一个内嵌的while循环。如果读入字符‘*’后再次读取的字符是‘/’则完成多行注释的识别，否则仍看作多行注释内部的内容，继续前面的处理，这是一个简单的if控制语句。实现代码如下：

```
1  case '/':
2      scan();
3      if(ch=='/'){                                     //单行注释

4          while(!(ch=='\n' || ch== -1))                //不是换行
5              scan();                                  符、文件结束符
6          t=new Token(ERR);
7      }
8      else if(ch=='*'){                                //多行注
```

释

```
9          while(!scan(-1)){                                //一直扫
描, 可能到文件结束

10          if(ch=='*'){                                      //出现
*
11          while(scan('*'));                                //跳过注
释内连续的
*
12          if(ch=='/'){                                       //多行注
释结束

13          t=new Token(ERR);
14          break;
15          }
16          }
17          }
18          if(!t&&ch==-1){                                    //未正常结束注释

19          LEXERROR(COMMENT_NO_END);
20          t=new Token(ERR);
21          }
22      }
23      else
24          t=new Token(DIV);
25      break;
```

第3~7行处理单行注释的识别，只要不是换行符或文件结束符，就一直扫描。

第8~22行处理多行注释的识别，第10~16行处理多行注释内出现字符‘*’的情况，第12~15行处理多行注释的结尾，第18~21处理多行注释在遇到‘*/’时文件结束的情况。

第24行处理除法运算符的识别。

前面提到，对于无效的词法记号，词法分析器有两种处理方式：不做任何处理，或返回错误词法记号**err**。如果采取不做任何处理的方

式，就需要在词法分析器内将错误词法记号忽略掉。具体的实现代码如下：

```
1  for(;ch!=-1;){                                //开始识别词法记号
2      Token*t=NULL;                              //词法记号指针
3      while(ch==' ' || ch=='\n' || ch=='\t')      //忽略空白符
4          scan();
5                                          //处理其他词法记号的识别
6      if(token)delete token;                    //删除旧的词法记号
7      token=t;                                  //记录新的词法记号
8      if(token&&token->tag!=ERR)                //有效词法记号
9          return token;                          //返回词法记号
10     else
11         continue;                              //继续识别新的词法记号
12 }
```

我们将词法记号的识别过程放在for循环内，循环终止条件是遇到文件结束符。

第2行使用t记录创建的词法记号对象指针。

第3~4行用于跳过空白字符。

第6~11行对产生的词法记号进行处理。第6~7行将当前新建的词法记号对象指针保存在token中。第8行如果判断出当前创建的词法记号是

错误词法记号，则继续识别新的词法记号，忽略错误词法记号。否则返回有效的词法记号，传递给语法分析器。这样，对于语法分析器来说，是不会接收到错误的词法记号的。

3.1.5 错误处理

在词法分析的过程中会出现词法错误的情况，需要进行错误处理。词法分析的错误处理只需要报告相应的词法错误信息，并给出错误出现的行号和列号。在我们实现的词法分析器中处理的词法错误如表3-4所示。

表3-4 词法错误表

词法错误类型	词法错误类型
STR_NO_R_QUTION	字符串丢失右引号
NUM_BIN_TYPE	二进制数没有实体数据
NUM_HEX_TYPE	十六进制数没有实体数据
CHAR_NO_R_QUTION	字符丢失右单引号
CHAR_NO_DATA	不支持空字符
OR_NO_PAIR	错误的“或”运算符
COMMENT_NO_END	多行注释没有正常结束
TOKEN_NO_EXIST	词法记号不存在

在扫描器中计算字符的行和列的位置，且保存处理的源文件名。根据表3-4提供的词法错误信息，很容易完成具体位置的词法错误信息输出。相关实现代码如下：

```
1  /*
2  词法错误类型

3  */
4  enum LexError
5  {
    STR_NO_R_QUTION, //字符串没有右引号
```

```

6      NUM_BIN_TYPE,                                //二进制数没有实体数
据

7      NUM_HEX_TYPE,                                //十六进制数没有实体
数据

8      CHAR_NO_R_QUTION,                            //字符没有右引号

9      CHAR_NO_DATA,                                //字符没有数据

10     OR_NO_PAIR,                                  //|| 只有一个
|
11     COMMENT_NO_END,                              //多行注释没有正常结
束

12     TOKEN_NO_EXIST                               //不存在的词法记号

13 };
14 /*
15 打印词法错误

16 */
17 void Error::lexError

(int code){
18 static const char *lexErrorTable[]={              //词法错误信息串

19     "字符串丢失右引号

",
20     "二进制数没有实体数据

",
21     "十六进制数没有实体数据

",
22     "字符丢失右单引号

",
23     "不支持空字符

",
24     "错误的

"或

"运算符

",
25     "多行注释没有正常结束

",
26     "词法记号不存在

"
27 };
28 errorNum++;

```

```
29 printf("%s<%d行  
,%d列  
> 词法错误  
: %s.\n",  
30 scanner->getFile(),  
31 scanner->getLine(),  
32 scanner->getCol(),  
33 lexErrorTable[code]  
34 );  
35 }  
36 #define LEXERROR  
(code) Error::lexError(code)
```

第1~13行使用枚举类型**LexError**记录所有词法错误的类型。

第14~35行定义输出词法错误信息的函数**lexError**，其中数组**lexErrorTable**保存与词法错误类型对应的信息。第28行使用变量**errorNum**记录编译器产生的错误数。第29~34行调用扫描器的方法获取词法错误产生位置所在的文件名、行号和列号。

第36行使用宏**LEXERROR**封装**lexError**函数的调用。

至此，我们详细介绍了词法分析器的所有实现细节，接下来进行语法分析时，只需调用词法分析器的**tokenize**函数，便可以获得源文件内定义的所有词法记号。

3.2 语法分析

语法分析器获取词法分析器提供的线性词法记号序列，根据高级语言文法结构，识别不同的语法模块。图3-13描述了语法分析器的结构。

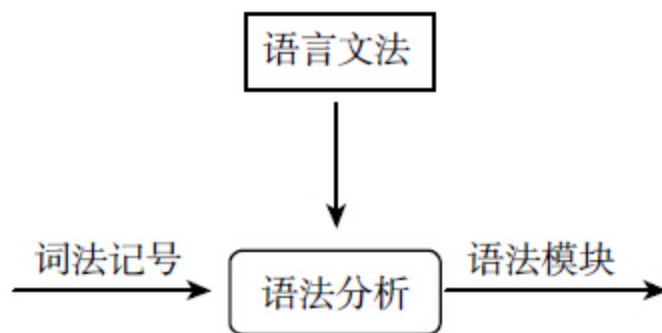


图3-13 语法分析器结构

经过语法分析器的处理后，高级语言源代码在编译器内部表现为一棵完整的抽象语法树，抽象语法树的子树（包括抽象语法树本身）也称为语法模块。高级语言的文法直接影响语法分析器的结构，在介绍语法分析器的构造之前，需要明确高级语言的文法定义。

3.2.1 文法定义

Chomsky于1956年建立了形式语言的描述，他把文法分为四种类型，即0型、1型、2型、3型，这四种文法描述的语言范围是依次缩减的。其中3型文法（即正则文法）描述了正则语言，前面讨论的词法记号属于3型文法，使用有限自动机可以完成正则语言的识别。而2型文法（即上下文无关文法）描述了程序设计语言，需要使用文法分析算法完成程序设计语言的识别。

在编译原理的教材中，描述了大量的文法分析算法。包括自顶向下的LL（1）分析、自底向上的LR分析等。LL（1）分析对文法的要求比LR分析严格，它不允许文法中的产生式出现左公因子和左递归，因此构造LL（1）文法时需要避免出现左公因子和左递归。虽然LR分析对文法的要求比较宽松，但是LR分析器手工构造较为复杂。鉴于LL（1）文法足以描述本文所实现的语言，本书采用LL（1）分析器分析自定义语言的文法。主流的编译器GCC也是使用LL分析器完成C语言的语法分析，不过GCC使用的是LL（2）分析算法。

我们知道，词法分析器将高级语言源代码转化为线性的词法记号序列，从语法分析器的角度来看，高级语言程序是由词法记号序列组合而成。在语法分析过程中，词法记号被称为终结符。将一组词法记

号子序列表示的抽象含义独立出来，使用符号表示，这些抽象符号被称为非终结符。

如果使用<type>表示自定义语言的数据类型，根据本书对自定义语言特性的定义，数据类型包含int、char和void三种基本类型，因此使用产生式表示为：

```
<type>->KW_INT | KW_CHAR | KW_VOID
```

其中KW_INT、KW_CHAR、KW_VOID三个词法记号分别对应int、char、void关键字词法记号，称为终结符。<type>作为这三个词法记号的抽象含义，被称为非终结符。产生式的含义为非终结符<type>可以推导出终结符KW_INT、KW_CHAR或KW_VOID。构造文法的过程其实就是对高级语言结构逐层解析的过程，接下来先根据自定义语言的特性，详细解析文法的构造过程。

1.高级语言程序

本书设计的自定义语言是C语言的子集，因此可以参考C语言的语法结构。C语言程序一般由变量声明、变量定义、函数声明、函数定义组合而成，如果使用非终结符<program>表示高级语言程序，使用<segment>表示组成程序的片段。那么使用产生式表示高级语言程序与程序片段的关系如下：

$\langle \text{program} \rangle \rightarrow \langle \text{segment} \rangle \langle \text{program} \rangle$

通过递归形式的定义， $\langle \text{program} \rangle$ 可以推导出任意多个 $\langle \text{segment} \rangle$ ，正好表示高级语言程序由任意多个程序片段组成的含义。但是高级语言程序包含的程序片段肯定是有限多个，上述推导的结果是无限多个程序片段，因此我们需要给推导过程一个终止条件。

$\langle \text{program} \rangle \rightarrow \epsilon$

我们使用特殊终结符 ϵ 表示空的终结符，只要 $\langle \text{program} \rangle$ 的推导过程中使用该条产生式，便可以终止递归推导的过程。同一个非终结符的产生式可以合并，使用符号 $' \mid '$ 连接产生式的右侧部分。

$\langle \text{program} \rangle \rightarrow \langle \text{segment} \rangle \langle \text{program} \rangle \mid \epsilon$

消除左递归

细心的读者会发现，既然可以使用递归形式的产生式表示“任意多个”的含义，那么上述产生式也可以表示为如下形式：

$\langle \text{program} \rangle \rightarrow \langle \text{program} \rangle \langle \text{segment} \rangle \mid \epsilon$

这样的产生式称为左递归形式，前面给出的是右递归形式，它们表示的含义完全等价。但是LL（1）文法不允许产生式出现左递归，因此我们应该尽可能消除左递归，左递归消除规则为：

对于包含左递归的产生式 $S \rightarrow Sa|b$ （其中大写字母表示非终结符，小写字母表示终结符），引入新的非终结符 S' ，将原产生式改写为：

```
S -> bS'  
S' -> aS' | ε
```

因此对于 $\langle \text{program} \rangle$ 的左递归定义 $\langle \text{program} \rangle \rightarrow \langle \text{program} \rangle$
 $\langle \text{segment} \rangle | \epsilon$ ，改写后的形式为：

```
<program> -> ε  
<program'>  
<program'> -> <segment><program'> | ε
```

产生式 $\langle \text{program} \rangle \rightarrow \epsilon \langle \text{program}' \rangle$ 等价于 $\langle \text{program} \rangle \rightarrow \langle \text{program}' \rangle$ ，这条产生式是冗余的，可以消除。使用终结符 $\langle \text{program} \rangle$ 代替 $\langle \text{program}' \rangle$ ，即得到最初的产生式 $\langle \text{program} \rangle \rightarrow \langle \text{segment} \rangle \langle \text{program} \rangle | \epsilon$ 。

将高级语言程序 $\langle \text{program} \rangle$ 拆分为程序片段 $\langle \text{segment} \rangle$ 后，对 $\langle \text{segment} \rangle$ 进行拆分。程序片段包含变量声明、变量定义、函数声明和函数定义，代码示例如下：

```

extern int name;                                //变量
name声明

extern int name();                              //函数
name声明

int name;                                       //变量
name定义

int *name;                                     //指针变量
name定义

int name(){}                                  //函数
name定义

int name();                                    //函数
name声明

```

可见，使用**extern**关键字引导的代码肯定是声明，紧跟**extern**之后的是数据类型，否则引导的代码可能是定义，也可能是函数声明。我们使用如下产生式表示<segment>。

```

<segment>->KW_EXTERN <segment'> | <segment'>
<segment'>-><type><def>
<type>->KW_INT | KW_CHAR | KW_VOID

```

根据是否包含**extern**将<segment>分为两类，并使用<segment'>表示公共的部分。公共的部分一般由类型非终结符开始，后面可能是变量名、函数名或指针变量，我们使用<def>作为它们的统称。

由于<segment'>只有一种推导方式，因此可以将之合并到<segment>的产生式内。

```
<segment>->KW_EXTERN <type><def> | <type><def>
<type>->KW_INT | KW_CHAR | KW_VOID
```

提取左公因子

或许读者好奇为什么大费周章地将<segment>做如此的拆分，而不是直接使用产生式表示它的结构，比如。

```
<segment>->KW_EXTERN KW_INT <def>
              | KW_EXTERN KW_CHAR <def>
              | KW_EXTERN KW_VOID <def>
```

我们发现，如此定义的<segment>的产生式右侧出现了相同的词法记号KW_EXTREN，称为左公因子。LL（1）文法通过超前查看一个词法记号来选择具体使用哪个产生式作为下一步的推导，左公因子的出现导致无法做出唯一的选择，因此需要提取左公因子。

对于包含左公因子的产生式“S->aA|aB”，引入新的非终结符S'，将原产生式改写为：

```
S->aS'
S'->A | B
```

此时A和B也不能出现左公因子，否则继续按照上述过程改写。将前面<segment>产生式改写后，得到：

```
<segment>->KW_EXTERN <segment'>
<segment'>->KW_INT <def> | KW_CHAR <def> | KW_VOID <def>
```

非终结符<segment'>的每个产生式右侧的首部可以合并为非终结符<type>，因此得到：

```
<segment'>-><type><def>
<type>->KW_INT | KW_CHAR | KW_VOID
```

这样就回到前面对<segment>定义的过程。

这里有一个细节需要注意，根据<segment>的产生式定义，发现如下特殊代码是可以被<segment>识别的。

```
extern int name(){}

```

这种由extern关键字引导的函数name的定义是符合我们定义的文法规则的，但这不是合法的C语言代码，按照我们对文法的定义是不能发现这个问题的。虽然可以通过不同的<def>形式来完成这种区分，但是这样做让文法显得更加复杂。对于这种情况，可以将这种合法性检查推迟到语义分析时进行。语义分析时，针对带有extern关键字的函数定义代码，会报告语义错误。这也说明了一点，在语法分析过程中难以或者无法处理的问题，可以由语义分析来辅助解决。

完成非终结符<segment>的拆分后，接下来是继续拆分非终结符<def>的结构。

2.声明与定义

非终结符<def>表示了变量函数的定义结构和不带extern关键字的函数声明，以下是满足<def>定义的示例代码。

```
int name;                                //未初始化变量
name定义

int name=1;                              //常量初始化变量
name定义

int name=a+b;                            //表达式初始化变量
name定义

int name,name2;                          //变量定义列表

int *ptr,arr[10];                         //指针
ptr和数组
arr定义

void fun();                              //函数
fun声明

void fun(){}                             //函数
fun定义

int fun(int x[10],char* y);              //有参数的函数
fun声明
```

满足<def>的代码非常多，如何进行合理的划分很关键。首先考虑变量的定义，按照引导词法记号可以将变量分为两类：以标识符ID开始的变量或数组和以词法记号MUL开始的指针。其中变量和数组的定义包含左公因子ID，因此需要提取左公因子。变量和指针是可以使用表达式初始化的（独立的常量、变量也是表达式），为了简化起见，不允许对数组初始化。使用<defdata>表示所有变量的定义，则产生式表示为：

```
<defdata>->ID <varrdef> | MUL ID <init>
<varrdef>->LBRACK NUM RBRACK | <init>

<init>->ASSIGN <expr> | ε
```

非终结符<varrdef>表示变量和数组的定义，<init>表示初始化部分，<expr>表示表达式，其定义在后面具体描述。

由于我们允许使用变量定义列表定义多个变量，除了第一个定义的变量外，后继的变量定义都是以', '进行分隔，并且可能出现任意多次。使用非终结符<deflist>表示除了第一个变量定义外剩余的部分，其产生式如下：

```
<deflist>->COMMA <defdata><deflist> | SEMICON
```

这里<deflist>仍使用左递归定义表示任意多个“COMMA<defdata>”的组合，并使用分号词法记号SEMICON终止左递归的无限推导过程。

这样一来，使用非终结符组合“<defdata><deflist>”便可以表示所有的变量定义结构。

再考虑函数的定义和声明。函数的定义和声明拥有公共的首部，包含函数名、左括号、形式参数列表和右括号。使用<fun>表示函数的定义和声明（除去返回类型的部分），使用<para>表示形式参数列表，使用<block>表示函数体，于是有：

```
<fun>->ID LPAREN <para> RPAREN SEMICON  
      | ID LPAREN <para> RPAREN <block>
```

提取左公因子后得到：

```
<fun>->ID LPAREN <para> RPAREN <funtail>  
<funtail>->SEMICON | <block>
```

那么对于<def>的定义，可以描述为：

```
<def>-><defdata><deflist> | <fun>
```

然而，这个产生式是不满足LL（1）文法的，因为非终结符<defdata>和<fun>有公共的左公因子ID！为了解决这个问题，需要将

<def>展开。

```
<def>->ID <varrdef><deflist>  
      | MUL ID <init><deflist>  
      | ID LPAREN <para> RPAREN <funtail>
```

然后合并包含左公因子ID的产生式，完成<def>的拆分。

```
<def>->ID <idtail> | MUL ID <init><deflist>  
<idtail>-><varrdef><deflist> | LPAREN <para> RPAREN <funtail>  
<funtail>->SEMICON | <block>
```

根据<def>的定义，函数的返回值类型只能是基本类型，而不能是指针类型，这是本书对文法的简化。另外，文法定义中允许出现void类型的变量，这个问题也留待语义分析时解决。

3.函数

函数声明和定义中使用<para>表示形式参数列表，函数定义中使用<block>表示函数体的内容。

为了简化文法的构造，对于形式参数的定义及其类型，我们做了如下几点限制：

- 1) 形式参数名称不能省略。
- 2) 形式参数不允许有默认值。

3) 数组参数必须指定数组长度。

除此之外，形式参数和变量定义的文法基本一致。使用<paradata>表示一个形式参数去除类型的部分，其产生式为：

```
<paradata>->MUL ID | ID <paradatatail>
```

```
<paradatatail>->LBRACK NUM RBRACK | ε
```

于是，使用非终结符组合“<type><paradata>”便可以表示一个完整的形式参数。

形式参数列表包含一个形式参数以及后继的以', '分割的任意多个形式参数的组合。使用非终结符<paralist>表示形式参数列表除了第一个形式参数之外的部分，使用左递归形式定义的产生式如下：

```
<paralist>->COMMA <type><paradata><paralist> | ε
```

使用非终结符组合“<type><paradata><paralist>”便可以表示完整的形式参数列表，当然，形式参数也可以为空。

```
<para>-><type><paradata><paralist> | ε
```

接下来拆分函数体<block>。函数体是由花括号包含起来的局部变量定义或语句的组合，因此<block>的产生式如下：

```
<block>->lbrac<subprogram>rbrac
<subprogram>-><localdef><subprogram>

|<statement><subprogram>

| ε
```

非终结符<subprogram>表示子程序的内容，<localdef>表示局部变量定义，<statement>表示语句。这里需要留意<localdef>和<statement>是否有左公因子，由于<localdef>都是以类型词法记号引导的，而<statement>不存在以类型词法记号开始的情况，因此不存在左公因子。

局部变量定义和前面描述的全局变量的定义完全相同，因此其产生式为：

```
<localdef>-><type><defdata><deflist>
```

由于自定义语言支持C语言常用的语句，因此语句的文法定义比较复杂。因为<statement>中包含了表达式，所以我们先描述表达式的文法，然后再讨论<statement>的文法定义。

这里从一般形式的表达式开始讨论表达式的文法。使用符号“⊕”表示通用的表达式运算符，使用小写字母表示表达式的操作数，那么表达式有如下形式：

```
a⊕  
b;  
a⊕  
b⊕  
c;  
a⊕  
b⊕  
c⊕  
d;
```

可见，表达式的文法也是使用递归的产生式表示：

```
<expr>-><operand><exprtail>  
<exprtail>-><operator><operand><exprtail> | SEMICON  
<operator>->⊕  
  
<operand>->a | b | c .. | z
```

对于表达式“a+b*c;”，经上述文法处理后，得到的抽象语法树形式如图3-14所示。

抽象语法树的结构实际上是对产生式展开后的形式，叶子节点表示终结符，非叶子节点表示非终结符，父节点到子节点的展开表示一次产生式的推导过程。在非终结符<exprtail>上，我们进行表达式的语义解析过程。在第一级<exprtail>子树处，根据读取的左操作数'a'、运

算符'+'和右操作数'b'构建表达式“a+b”，结果保存到临时变量't1'。在第二级<exprtail>子树处，根据读取的左操作数't1'、运算符'*'和右操作数'c'构建表达式“t1*c”，结果保存到临时变量't2'中。在第三级<exprtail>子树处，读取操作数't2'，此处不再有新的运算符和操作数，因此不构建任何表达式。

我们发现，按照抽象语法树的解析方式，表达式“a+b*c”实际被解析为“（a+b）*c”，这错误地解析了表达式的原有含义！而按照运算符优先级的规定，原表达式应该被解析为“a+（b*c）”。因此，使用通用的运算符的概念构造表达式文法并不可靠，我们需要在文法内反映出运算符的优先级特性。

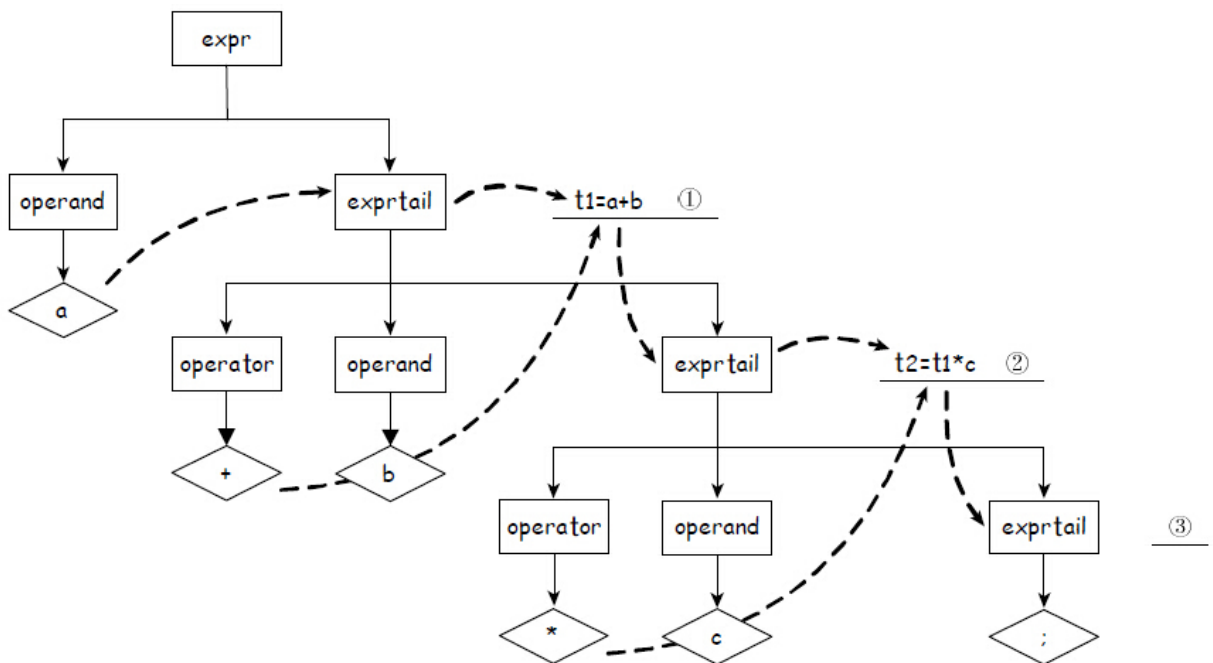


图3-14 表达式“a+b*c”抽象语法树

构造保持运算符优先级特性的文法的方法是：将高优先级运算符形成的表达式整体作为低优先级运算符形成的表达式的操作数。

按照运算符优先级构建表达式文法，识别表达式“a+b*c”的抽象语法树形式如图3-15所示。

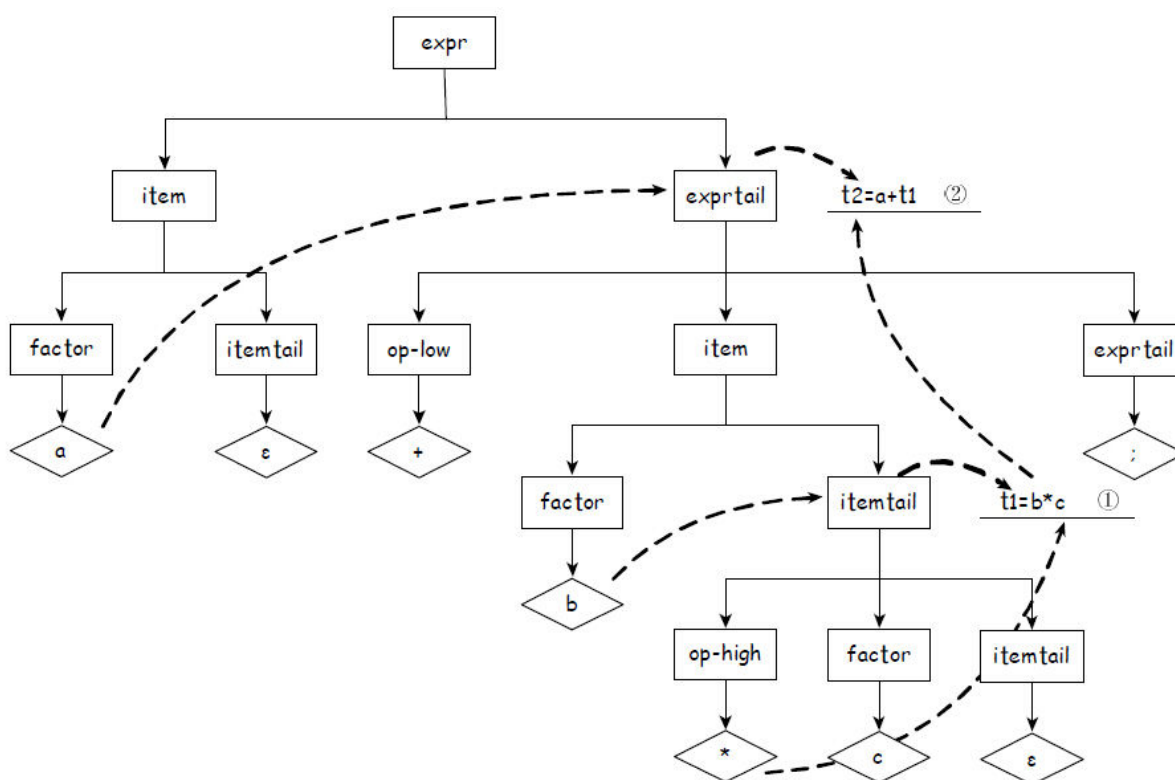


图3-15 考虑运算符优先级的表达式“a+b*c”抽象语法树

考虑运算符的优先级后，运算符‘*’的表达式子树优先被处理。在非终结符<item>处，根据读取的左操作数‘b’、运算符‘*’和右操作数‘c’构建表达式“b*c”，结果保存到临时变量‘t1’中。在非终结符<exprtail>处，根据读取的左操作数‘a’、运算符‘+’和右操作数‘t1’构建表达式“a+t1”，结果

保存到临时变量't2'中。通过这种方式可以正确解析表达式“a+b*c”的原有含义。

根据运算符的优先级，我们重新构造表达式的文法。

```
<expr>-><item><exprtail>
<exprtail>-><op-low><item><exprtail> | SEMICON
<op-low>->+
<item>-><factor><itemtail>
<itemtail>-><op-high><factor><itemtail> | ε

<op-high>->*
<factor>->a | b | c .. | z
```

从前面的讨论可以看出，运算符的优先级影响表达式的文法，那么运算符的结合性是否也对表达式文法有影响呢？回顾图3-15表达式的抽象语法树，我们发现在非终结符<exprtail>或<itemtail>处，可以进行灵活的选择。例如对于非终结符<exprtail>，可以选择：

1) 按照运算符<op-low>将右操作数<item>与前面的左操作数<item>结合，再处理后面的表达式<exprtail>。这种方式称为运算符的左结合，比如表达式“a+b+c”被处理为“(a+b) +c”。

2) 先按照<exprtail>提供的运算符，将右操作数<item>与后面的表达式<exprtail>进行结合，再按照运算符<op-low>，将得到的结果与前面的左操作数<item>进行结合。这种方式称为运算符的右结合，比如表达式“a=b=c”被处理为“a= (b=c) ”。

因此，运算符的结合性不影响表达式文法的构造。通过对表达式语义不同方式的处理，可以正确地表达运算符的结合性，在后面代码生成部分会对其做详细的描述。

表3-5给出了所有运算符的优先级和结合性，其中运算符的优先级值越小，优先级越高。

表3-5 运算符优先级与结合性

运算符	含 义	优先级	结合性
=	赋值	10	右结合
	逻辑或	9	左结合
&&	逻辑与	8	左结合
> < >= <= == !=	大于、小于、大于等于、小于等于、等于、不等于	7	左结合
+ -	加法、减法	6	左结合
* / %	乘法、除法、取模	5	左结合
! - & * ++ --	逻辑非、取负、取址、指针、前置++、前置--	4	右结合
++ --	后置++、后置--	3	右结合
()	括号	2	左结合
[] ()	数组索引、函数调用	1	左结合

根据运算符的优先级，可以按照前面讨论的方法构造自定义语言表达式的文法。从最低优先级的赋值运算符到最高优先级运算符的表达式文法的构造方式如下。

赋值表达式的运算符为**ASSIGN**，包含两个逻辑“或”表达式操作数。赋值表达式<assexpr>的文法为：

```
<assexpr>-><orexpr><asstail>

<asstail>->ASSIGN <orexpr><asstail> | ε
```

严格来说，赋值运算符的左操作数只能是左值表达式。逻辑“或”表达式一定是右值表达式，是不能作为赋值运算符的左操作数的。这个问题也滞后到语义分析时进行处理。

逻辑“或”表达式的运算符为OR，包含两个逻辑“与”表达式操作数。逻辑“或”表达式<orexpr>的文法为：

```
<orexpr>-><andexpr><ortail>

<ortail>->OR <andexpr><ortail> | ε
```

逻辑“与”表达式的运算符为AND，包含两个关系表达式操作数。逻辑“与”表达式<andexpr>的文法为：

```
<andexpr>-><cmpexpr><andtail>

<andtail>->AND <cmpexpr><andtail> | ε
```

关系表达式的运算符为GT|GE|LT|LE|EQU|NEQU，包含两个算术表达式操作数。关系表达式<cmpexpr>的文法为：

```
<cmpexpr>-><aloexpr><cmptail>

<cmptail>-><cmps><aloexpr><cmptail> | ε

<cmps>->GT | GE | LT | LE | EQU | NEQU
```

算术表达式的运算符为ADD|SUB，包含两个项表达式操作数。算术表达式<aloexpr>的文法为：

```
<aloexpr>-><item><alotail>

<alotail>-><adds><item><alotail> | ε
<adds>->ADD | SUB
```

项表达式的运算符为MUL|DIV|MOD，包含两个因子表达式操作数。项表达式<item>的文法为：

```
<item>-><factor><itemtail>

<itemtail>-><muls><factor><itemtail> | ε
<muls>->MUL | DIV | MOD
```

因子表达式的运算符为NOT|SUB|LEA|MUL|INCR|DECR，包含一个仍为因子表达式的操作数，因子表达式可以是值表达式。因子表达式比较特殊，运算符出现在操作数的左侧。因子表达式<factor>的文法为：

```
<factor>-><lop><factor>|<val>

<lop>->NOT | SUB | LEA | MUL | INCR | DECR
```

值表达式的运算符为INCR|DECR，包含一个元素表达式操作数。值表达式比较特殊，运算符出现在操作数的右侧，且只能出现一次。引入值表达式主要是方便循环语句内循环因子的自加或自减。值表达式<val>的文法为：

```
<val>-><elem><rop>
<rop>->INCR | DECR
```

元素表达式不包含运算符，它是表达式的基本操作数单元。就我们所知，可以参与表达式运算的操作有变量、数组、函数调用、括号表达式和常量。元素表达式<elem>的文法为：

```
<elem>->ID
      | ID LBRACK <expr> RBACK
      | ID LPAREN <realarg> RPAREN
      | LPAREN <expr> RPAREN
      | <literal>
```

其中前三条产生式包含左公因子ID，提取左公因子后得到：

```
<elem>->ID <idexpr> | LPAREN <expr> RPAREN | <literal>

<idexpr>->LBRACK <expr> RBRACK | LPAREN <realarg> RPAREN | ε
```

函数调用的实际参数列表是由逗号分割的表达式列表或为空，文法如下：

```
<realarg>-><arg><arglist> |  $\epsilon$   
  
<arglist>->COMMA <arg><arglist> |  $\epsilon$   
  
<arg>-><expr>
```

常量包含数字、字符和字符串:

```
<literal>->NUM | CH | STR
```

这样，我们完成了赋值表达式的文法构造。由于没有比赋值运算符更低级的运算符，因此表达式直接用赋值表达式表示:

```
<expr>-><assexpr>
```

考虑循环语句内循环条件使用的表达式可以为空，因此我们使用非终结符<altexpr>表示可以为空的表达式。

```
<altexpr>-><expr> |  $\epsilon$ 
```

至此，我们完成了表达式文法的构造。

4.语句

完成表达式的文法构造后，语句的文法构造就比较容易了。自定义语言包含的语句有表达式语句；while、do-while、for循环语句；if-else、switch-case分支语句，以及break、continue和return语句。

语句的文法定义如下：

```
<statement>-><altexpr>SEMICON  
  
    | <whilestat>|<forstat>|<dowhilestat>  
  
    | <ifstat>|<switchstat>  
  
    | KW_BREAK SEMICON  
  
    | KW_CONTINUE SEMICON  
  
    | KW_RETURN <altexpr> SEMICON
```

while循环语句的文法如下：

```
<whilestat>->KW_WHILE LPAREN <altexpr> RPAREN <block>
```

其中<altexpr>允许循环条件为空，空循环条件表示永真。<block>表示循环体，与函数体内容等价。

do-while循环语句的文法为：

```
<dowhilestat>->KW_DO <block> KW_WHILE LPAREN <altexpr> RPAREN SEMICON
```

for循环语句的文法为:

```
<forstat>->KW_FOR LPAREN <forinit><altexpr> SEMICON <altexpr> RPAREN <block>  
<forinit>-><localdef> | <altexpr> SEMICON
```

其中<forinit>表示for循环的初始化语句，它可以是局部变量的定义，也可以是表达式语句。

if-else分支语句的文法为:

```
<ifstat>->KW_IF LPAREN <expr> RPAREN <block><elsestat>  
  
<elsestat>->KW_ELSE <block> |  $\epsilon$ 
```

其中if语句的条件表达式不允许为空，因此使用<expr>表示，而不是<altexpr>，else语句可以不存在。

switch-case分支语句的文法为:

```
<switchstat>->KW_SWITCH LPAREN <expr> RPAREN LBRAC <casestat> RBRAC  
<casestat>->KW_CASE <caselabel> COLON <subprogram><casestat>  
  
| KW_DEFAULT COLON <subprogram>  
<caselabel>-><literal>
```

其中，非终结符<casestat>表示多个case语句，且以default语句结束。<caselabel>表示case的标签，必须是数字或字符常量，而不能是字符串常量，这个问题在需要语义分析时进行解决。

通过构造语句的文法，我们可以得出一个结论：表达式为程序提供真正的计算，语句为程序提供控制流程，函数为程序提供功能封装，全局变量为程序提供信息共享。

至此，我们完成了所有文法的定义。接下来，就是根据文法定义构建语法分析器，识别程序的语法模块。

3.2.2 递归下降子程序

前面讨论过，我们使用LL（1）文法分析算法可以完成高级语言的语法分析。一般编译原理教材中，描述了通过计算LL（1）文法的FIRST、FOLLOW和SELECT集合，构建LL（1）分析表进行语法分析，这有点类似基于表驱动的词法分析。本书不讨论LL（1）分析表的构建方法，对此感兴趣的读者可以参考编译原理相关教材。我们仍采用“硬编码”的方式进行LL（1）分析，即递归下降子程序。

递归下降子程序实现的语法分析器是由一系列的子程序完成的，不同的子程序负责识别不同的语法模块。由于语法模块与抽象语法子树一一对应，且抽象语法子树是由非终结符通过产生式展开形成，因此每个非终结符与子程序一一对应。子程序有可能是递归的，说明子程序可以完成重复形式的语法模块识别，这与右递归形式的产生式是对应的。从这一点也能说明LL（1）文法的产生式为何不能出现左递归，因为左递归的子程序无法终止。从抽象语法树的形式上看，递归下降子程序是从树根的非终结符开始，依次展开直达叶子节点，是一个自顶向下的过程。抽象语法树的展开由产生式的推导形成，产生式右侧的终结符直接与输入的词法记号进行匹配，产生式右侧的非终结符转化为对应子程序的函数调用。

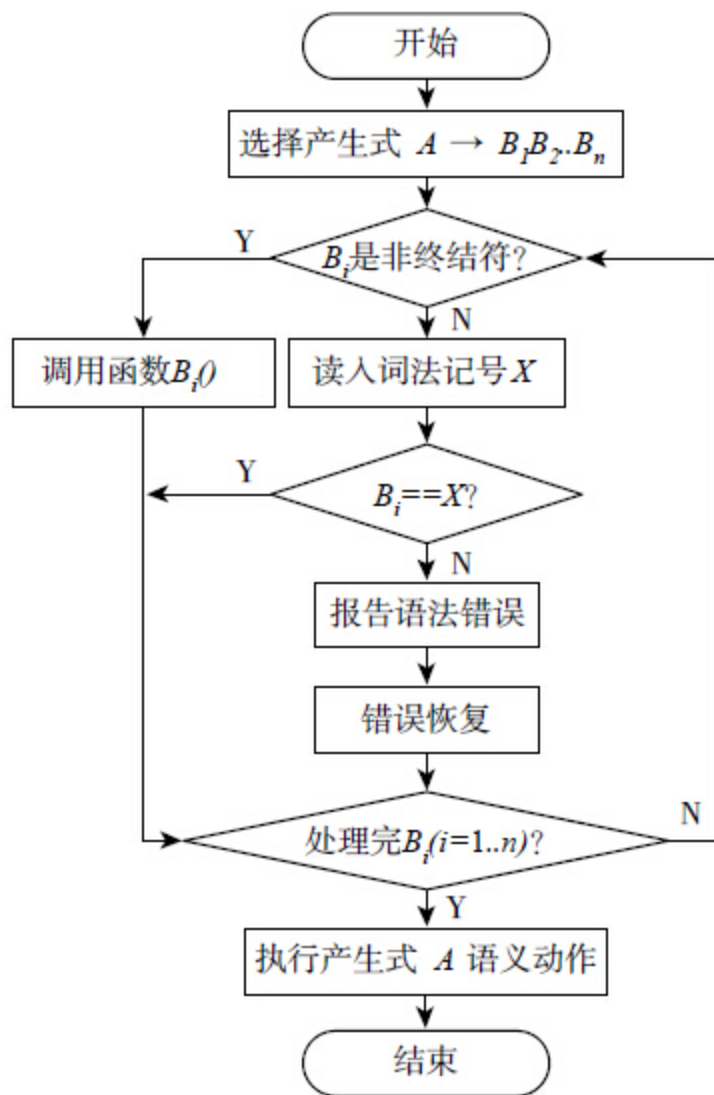


图3-16 递归下降子程序的构造

图3-16描述了递归下降子程序的构造方法，其构造规则描述如下：

- 1) 文法中的每个非终结符对应一个子程序（函数）。
- 2) 文法中的每个产生式对应子程序内的一个分支。

- 3) 产生式中的非终结符转化为对应子程序的调用。
- 4) 产生式中的终结符与当前读入的词法记号进行匹配。
- 5) 终结符与当前读入的词法记号匹配失败时，需要报告语法错误，并进行相应的错误修复。
- 6) 产生式处理时，需要执行当前子程序分支相关的语义动作，比如符号表管理、语义分析和代码生成。

举例说明，例如while循环语句的文法。

```
<whilestat>->KW_WHILE LPAREN <altexpr> RPAREN <block>
```

按照上述规则构造<whilestat>的递归下降子程序为：

```
1 void Parser::whilestat
2 {
3     match(KW_WHILE
4 );
5     if(!match(LPAREN
6 ))
7         recovery(EXPR_FIRST|F(RPAREN)
8             ,LPAREN_LOST,LPAREN_WRONG);
9     altexpr
10 );
11     if(!match(RPAREN
12 ))
13         recovery(F(LBRACE),RPAREN_LOST,RPAREN_WRONG);
14     block
15 );
16 }
```

产生式左侧的非终结符<whilestat>转化为子程序whilestat。

第7、10行将非终结符<altexpr>和<block>转化为子程序altexpr和block的调用。

第3、4、8行使用match函数对终结符KW_WHILE、LPAREN、RPAREN进行匹配，match的函数功能与词法分析器的scan函数类似。

```
1 void Parser::move
  ()
2 {
3     look=lexer.tokenize();
4 }
5 bool Parser::match
  (Tag need)
6 {
7     if(look-
tag==need){
8         move();
9         return true;
10    }
11    else
12        return false;
13 }
```

其中move函数使用词法分析器的tokenize函数读取词法记号，并将之记录到变量look中（保存了当前读入的待匹配的词法记号）。match将参数need与读入的词法记号进行匹配，匹配成功后读入下一个词法记号，返回真。否则，返回假，不继续读入词法记号。

whilestat函数的第3行使用match函数匹配KW_WHILE时并未判断终结符是否匹配，这是因为<statement>产生式包含<whilestat>，statement子程序在调用whilestat之前已经判断了look是KW_WHILE，

因此match必然返回真。而对LPAREN和RPAREN则需要判断是否与look匹配，不匹配时则调用recovery函数报告语法错误并进行错误恢复。错误恢复算法的实现在后面会具体描述。

对于非终结符<whilestat>，它只包含一条产生式，因此子程序whilestat内只有一个分支流程。

```
<statement>-><altexpr>SEMICON
                | <whilestat>|<forstat>|<dowhilestat>
                | <ifstat>|<switchstat>
                | KW_BREAK SEMICON
                | KW_CONTINUE SEMICON
                | KW_RETURN <altexpr> SEMICON
```

而对于非终结符<statement>则包含多个产生式，其子程序statement包含多个分支。

```
1 void Parser::statement
2 {
3     switch(look->tag)
4     {
5         case KW_WHILE:whilestat
6         ();break;
7         case KW_FOR:forstat
8         ();break;
9         case KW_DO:dowhilestat
10        ();break;
11        case KW_IF:ifstat
12        ();break;
13        case KW_SWITCH:switchstat
14        ();break;
15        case KW_BREAK
16        :
17        move();
18        if(!match(SEMICON
19        ))
```

```

13             recovery(TYPE_FIRST||STATEMENT_FIRST||F(RBRACE)
14                       , SEMICON_LOST, SEMICON_WRONG);
15         break;
16     case KW_CONTINUE
17     :
18         move();
19         if(!match(SEMICON
20 ))
21             recovery(TYPE_FIRST||STATEMENT_FIRST||F(RBRACE)
22                       , SEMICON_LOST, SEMICON_WRONG);
23         break;
24     case KW_RETURN
25     :
26         move();
27         altexpr
28     ();
29         if(!match(SEMICON
30 ))
31             recovery(TYPE_FIRST||STATEMENT_FIRST||F(RBRACE)
32                       , SEMICON_LOST, SEMICON_WRONG);
33         break;
34     default:
35         altexpr
36     ();
37         if(!match(SEMICON
38 ))
39             recovery(TYPE_FIRST||STATEMENT_FIRST||F(RBRACE)
40                       , SEMICON_LOST, SEMICON_WRONG);
41     }
42 }

```

由于look保存的是当前读入的待匹配的词法记号，第3行根据look的tag字段对应的终结符选择相应的产生式。

第5~9行实现了以非终结符开始的产生式，case标签的值为该非终结符的FIRST集。假如某个非终结符的FIRST集合元素个数大于1，则应该使用if分支语句替换switch语句。

第10~33行实现了以终结符开始的产生式，case标签的值为产生式的第一个终结符。

从递归下降子程序的实现来看，可以发现构造LL（1）文法时提取左公因子的原因。这是因为产生式包含左公因子时，递归下降子程序无法通过条件判断选择唯一的分支流程。

回到文法定义的第一条产生式：

```
<program>-><segment><program> |  $\epsilon$ 
```

这种类型的产生式比较特殊，因为产生式内包含空终结符 ϵ 。空终结符不是有效的词法记号，因此不能通过对look的判断来决定不同产生式的选择。当产生式为空终结符时，使用产生式左侧的非终结符的FOLLOW集合作为选择产生式的条件。

SELECT集合

在编译原理教材中，有关LL（1）分析的章节中会讨论SELECT集合的概念，SELECT集合的定义如下：

给定产生式 $A \rightarrow a$ ，如果 a 不能推导出空终结符 ϵ ，则 $\text{SELECT}(A \rightarrow a) = \text{FIRST}(a)$ 。如果 a 可以推导出空终结符 ϵ ，则 $\text{SELECT}(A \rightarrow a) = (\text{FIRST}(a) - \{\epsilon\}) \cup \text{FOLLOW}(A)$ 。

而判定一个文法是否是LL (1) 文法的充要条件是：对任意非终结符A的任意两个不同的产生式A→a和A→b，满足SELECT (A→a) ∩ SELECT (A→b) = ∅ 。

因此，前面讨论了产生式递归下降子程序的方法正是SELECT集合特点的体现。即产生式右侧不能推导出空非终结符时使用产生式的FIRST集合决定产生式的选择。否则，还要考虑产生式的FOLLOW集合。只要满足LL (1) 文法，则保证同一个终结符的产生式的SELECT集合互不相交，从而保证了产生式选择的唯一性。

非终结符<program>表示整个源程序，其FOLLOW集合只包含一个非终结符——文件结束符词法记号END。因此，program子程序实现如下：

```
1 void Parser::program
2 {
3     if(look->tag==END
4         return ;
5     }
6     else{
7         segment
8         program
9     }
10 }
```

当遇到终结符END时，`program`子程序退出，停止右递归过程。这里可以看出LL（1）文法不允许出现左递归的原因，假如交换第7行与第8行的代码顺序，如果不考虑判断条件，`program`函数将是无限递归的过程。

由于`program`子程序是语法分析过程中最先执行的子程序，因此在执行`program`子程序之前，需要调用`move`函数将第一个词法记号读入变量`look`。

```
1 void Parser::analyze
  ()
2 {
3     move();
4     program();
5 }
```

函数`analyze`是语法分析器的主程序，调用它可以完成语法分析过程。

包含空终结符的产生式还有一种特殊情况，例如非终结符`<init>`的产生式：

```
<init>->ASSIGN <expr> |  $\epsilon$ 
```

该产生式除了产生空终结符之外，其他产生式都是以终结符开始的，我们优先对该类产生式进行判断选择。

```
1 void Parser::init
2 {
3     if(match(ASSIGN)){
4         expr();
5     }
6 }
```

当遇到终结符ASSIGN后，则调用expr子程序继续分析，否则结束init子程序。这里不需要对空终结符产生式进行处理，即不通过判断<init>的FOLLOW集来选择空终结符对应的产生式。因为后继的其他子程序读入这个词法记号时，会立即发现这个错误。

按照前面描述的递归下降子程序的构造方法，可以将3.2.1节定义的全部文法转化为语法分析程序，这里不再重复列举其他子程序的实现代码。我们发现，递归下降子程序实现的语法分析器并未像图3-13描述的那样输出语法模块的信息。编译原理的教材中一般将语法分析器的输出描述为抽象语法树，抽象语法树的每个节点也称为语法模块，对应递归下降子程序的每个子程序。我们可以为每个子程序添加抽象语法树生成代码，将抽象语法树保存到内存数据结构或者临时文件内。但是还有一种更简单的方式是直接在递归下降子程序内进行语义动作，这是因为递归下降子程序的递归调用过程已经蕴含了抽象语法树的结构。正如图3-16描述的那样，在子程序内执行符号表管理、语义分析和代码生成的工作。这样的过程称为语法制导，即根据语法分析识别的流程来确定语法模块，并进行相关的语义动作。

3.2.3 错误处理

对于合法的源程序输入，执行上述构造完成的语法分析程序后，`analyze`函数会正常结束。但是，一个健壮的语法分析器在语法分析出错时，要能恢复到正常语法分析流程。

例如，源程序为“`int a;` ”，通过词法分析产生的词法记号序列为（`KW_INT`，`ID`，`SEMICON`，`END`），语法分析可以正常分析这段程序。假如将源程序修改为“`a;` ”，词法分析器产生的词法记号序列为（`ID`，`SEMICON`，`END`）。而根据正常的语法分析流程，首先读入`ID`，并与类型`<type>`匹配，匹配失败报告类型匹配错误。然后读入`SEMICON`，并与`ID`匹配，匹配失败报告标识符匹配错误。最后读入`END`，并与`SEMICON`匹配，匹配失败报告分号丢失错误。我们发现，原本只是一个类型丢失的语法错误，按照上述语法分析过程，会导致后继终结符的匹配发生“连锁反应”，从而产生更多的语法错误。

如果语法分析器在识别出某个非终结符丢失错误时，及时修正词法记号读取的“位置”，将语法分析过程恢复到正常的流程上来，便可以有效避免语法错误的“连锁反应”。为此，我们设计了一个简单的语法错误恢复算法。

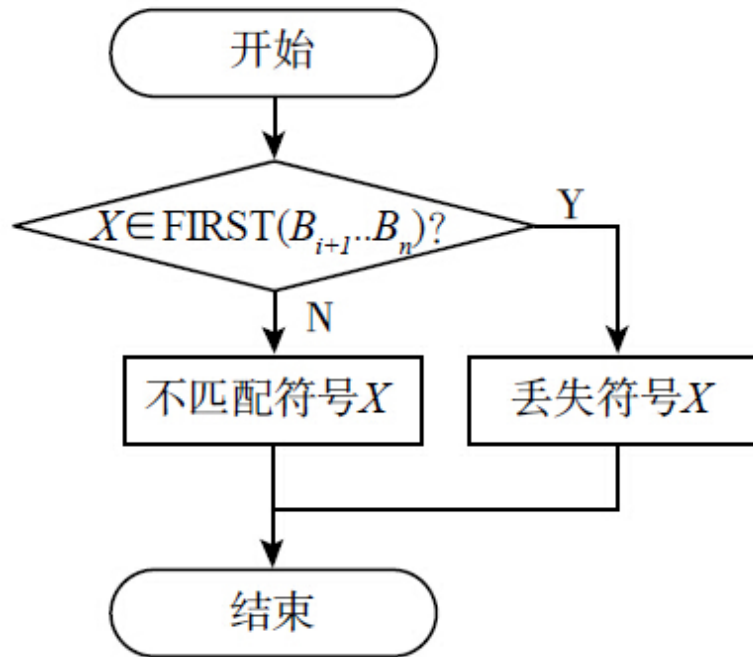


图3-17 错误恢复算法流程

图3-16中，在处理产生式 $A \rightarrow B_1 B_2 \dots B_n$ 时，如果在终结符 B_i 处与读入的词法记号 X 匹配失败，则进行如图3-17的错误恢复处理。通过判断读入的词法记号 X 是否属于终结符 B_i 后继产生式 $B_{i+1} \dots B_n$ 的FIRST集，来决定当前待匹配的非终结符是符号丢失错误，还是符号匹配错误。其中 $\text{FIRST}(B_{i+1} \dots B_n)$ 的定义为：

1) 如果 B_{i+1} 不能推导出空终结符 ϵ ，则 $\text{FIRST}(B_{i+1} \dots B_n) = \text{FIRST}(B_{i+1})$ 。

2) 如果 B_{i+1} 能推导出空终结符 ϵ ，则 $\text{FIRST}(B_{i+1} \dots B_n) = (\text{FIRST}(B_{i+1}) - \{\epsilon\}) \cup \text{FIRST}(B_{i+2} \dots B_n)$ 。

3) 对于 $\text{FIRST}(B_n)$ ，如果 B_n 能推导出空终结符 ϵ ，则 $\text{FIRST}(B_n) = (\text{FIRST}(B_n) - \{\epsilon\}) \cup \text{FOLLOW}(A)$ 。

根据这样的形式化定义， $\text{FIRST}(B_{i+1}..B_n)$ 集合内保存了终结符 B_i 后可能出现的所有终结符。图3-17描述的错误恢复算法的基本思想是，若终结符匹配失败，则检查当前读入的词法记号是否是待匹配非终结符之后可能出现的词法记号。如果是，则认为待匹配的非终结符丢失，否则认为待匹配的非终结符匹配出错，继续读入下一个词法记号。其代码描述如下：

```
1 void Parser::recovery
  (bool cond, SynError lost, SynError wrong)
2 {
3     if (cond)                                     //在给定的
Follow集合内

4         SYNERROR(lost, look);
5     else{
6         SYNERROR(wrong, look);
7         move();
8     }
9 }
```

函数`recovery`描述了错误恢复的算法实现，参数`cond`描述当前读入的词法记号`look`是否属于 $\text{FIRST}(B_{i+1}..B_n)$ 集合，而参数`lost`和`wrong`描述了未匹配终结符 B_i 的丢失和匹配错误信息的类型。`SYNERROR`宏用于输出语法错误信息，后面会详细介绍它的实现。

参数`cond`保存逻辑真假值，表示当前词法记号`look`是否属于 $\text{FIRST}(B_{i+1}..B_n)$ 集合，在我们实现的语法分析器中并未计算 $\text{FIRST}(B_{i+1}..B_n)$

.. B_n), 而是直接通过硬编码完成集合元素的比较。假如集合FIRST ($B_{i+1} .. B_n$) = {KW_INT, KW_CHAR, KW_VOID} (这里 B_i 是非终结符<type>), 那么计算cond值使用的代码为:

```
look->tag==KW_INT || look->tag==KW_CHAR || look->tag==KW_VOID
```

为了将代码简化, 我们定义如下两个简单的宏代替这个逻辑表达式。

```
#define _(T) ||look->tag==T  
#define F(C) look->tag==C
```

使用这两个宏计算cond值的逻辑表达式为:

```
F(KW_INT)_(KW_CHAR)_(KW_VOID)
```

使用逻辑表达式确定look是否属于FIRST ($B_{i+1} .. B_n$) 集合比构造该集合后再进行判断更加高效。

需要说明的是, 我们实现的错误恢复算法虽然可以在一定程度上避免因终结符丢失时产生的语法错误而引起的连锁反应, 但是仍有不足之处。例如终结符匹配失败时, 如果读入的词法记号不在FIRST

($B_{i+1} .. B_n$) 集合内, 那么再次读入的词法记号将仍会按照原来的语法分析流程继续分析, 而不能保证该词法记号的分析过程进入正确的语法分析器程序, 从而也可能报告更多的语法错误。实际针对语法分析

器的使用测试中，这样的情况出现的次数并不多，这是因为大部分语法分析出错的代码很多情况来源于编程者的书写错误和疏漏。即使语法分析器报告了大量的不该出现的语法错误信息，我们仍能确定第一条语法错误的准确性，通过对错误的修改，可以有效地减少类似错误情况的出现。当然，我们也可以对以上的错误恢复算法做一定的修改。在出现终结符匹配失败错误时，不是简单地读入下一个词法记号，而是尝试读入更多的词法记号，直到读入的词法记号在FIRST $(B_{i+1} \dots B_n)$ 集合为止。这样做看起来每次的错误恢复都能保证语法分析回到正常的流程，但是一旦出现将所有的词法记号读取完毕仍不能发现FIRST $(B_{i+1} \dots B_n)$ 集合内的元素时，就会产生更多的语法错误。可见，错误恢复算法并没有实际的标准，语法分析器的实现者需要根据具体需要做不同的选择。我们这里只是描述了错误恢复算法的基本思想，读者可以尝试编写自己的错误恢复算法，并对错误恢复算法的健壮性进行测试。

在报告词法错误的信息时，需要指定出现词法错误的位置，包括文件名、行列位置和错误类型信息。而在报告语法错误时，需要指定语法错误的位置，包括文件名、行位置、错误类型以及出错时读入的词法记号——它标识了语法错误的具体的行内位置。在我们实现的语法分析器中处理的语法错误如表3-6所示。

表3-6 语法错误表

符号类型	语法错误类型	
	符号丢失错误	符号匹配错误
类型	TYPE_LOST	TYPE_WRONG
标识符	ID_LOST	ID_WRONG
数字	NUM_LOST	NUM_WRONG
常量	LITERAL_LOST	LITERAL_WRONG
,	COMMA_LOST	COMMA_WRONG
;	SEMICON_LOST	SEMICON_WRONG
=	ASSIGN_LOST	ASSIGN_WRONG
:	COLON_LOST	COLON_WRONG
while	WHILE_LOST	WHILE_WRONG
(LPAREN_LOST	LPAREN_WRONG
)	RPAREN_LOST	RPAREN_WRONG
[LBRACK_LOST	LBRACK_WRONG
]	RBRACK_LOST	RBRACK_WRONG
{	LBRACE_LOST	LBRACE_WRONG
}	RBRACE_LOST	RBRACE_WRONG

在扫描器内，我们计算了字符所在行的位置，同时保存了处理的源文件的名称，语法分析器内look保存了当前读入的词法记号。根据表3-6提供的语法错误信息，实现语法错误信息输出的相关代码如下：

```

1  /*
2   语法错误类型

3  */
4  enum SynError

{
5      TYPE_LOST,                //类型

6      TYPE_WRONG,
7      ID_LOST,                  //标识符

8      ID_WRONG,
9      NUM_LOST,                //数组长度

10     NUM_WRONG,
11     LITERAL_LOST,            //常量

12     LITERAL_WRONG,

```

```

13      COMMA_LOST,                                //逗号

14      COMMA_WRONG,
15      SEMICON_LOST,                              //分号

16      SEMICON_WRONG,
17      ASSIGN_LOST,                               // =
18      ASSIGN_WRONG,
19      COLON_LOST,                                //冒号

20      COLON_WRONG,
21      WHILE_LOST,                                //while
22      WHILE_WRONG,
23      LPAREN_LOST,                               //(
24      LPAREN_WRONG,
25      RPAREN_LOST,                               //)
26      RPAREN_WRONG,
27      LBRACK_LOST,                               //[
28      LBRACK_WRONG,
29      RBRACK_LOST,                               //]
30      RBRACK_WRONG,
31      LBRACE_LOST,                               //{
32      LBRACE_WRONG,
33      RBRACE_LOST,                               //{
34      RBRACE_WRONG
35 };
36 /*
37 打印语法错误

38 */
39 void Error::synError

(int code,Token*t){
40 static const char *synErrorTable[]=
41 {
42     "类型

",
43     "标识符

",
44     "数组长度

",
45     "常量

",
46     "逗号

",
47     "分号

",
48     "=",
49     "冒号

",
50     "while",
51     "(",
52     ")",
53     "[",
54     "]",
55     "{",

```

```

56     "}"
57     };
58     errorNum++;
59     if(code%2==0)                                     //lost
60         printf("%s<第
%d行
> 语法错误
: 在
%s 之前丢失
%s .\n"
61         , scanner->getFile(), scanner->getLine()
62         , t->
toString().c_str(), synErrorTable[code/2]);
63     else                                             //wrong
64         printf("%s<第
%d行
> 语法错误
: 在
%s 处没有正确匹配
%s .\n"
65         , scanner->getFile(), scanner->getLine()
66         , t->toString().c_str(), synErrorTable[code/2]);
67 }
68 #define SYNERROR
(code, t) Error::synError(code, t)

```

第1~35行使用枚举类型**SynError**记录了所有语法错误的类型。

第36~67行定义输出语法错误信息的函数**synError**，其中数组**synErrorTable**保存了与语法错误类型对应的符号类型信息。第58行使用变量**errorNum**记录编译过程中产生的错误数。第59~66行调用了扫描器的方法以获取词法错误所在的文件名和行号，其中参数**t**记录了语法分析器当前读入的词法记号**look**。

第68行使用宏**SYNERROR**封装了**synError**函数的调用。

至此，我们介绍了语法分析器的所有实现细节。接下来是在递归下降子程序内插入基于语法制导的语义动作代码，包括符号表管理、语义分析和代码生成。

3.3 符号表管理

符号表内记录了编译过程中产生的关键信息，我们通过在语法分析程序插入符号表管理代码，进行变量信息管理、函数信息管理、作用域管理等工作。

如图3-18所示，符号表管理、语义分析和代码生成没有绝对的先后关系。语法分析中产生的语义动作除了更新符号表信息，也需要进行代码生成的工作。在符号表信息更新和代码生成过程中，语义分析需要检查代码语义信息的正确性。而语义分析和代码生成则需要从符号表内读取所需的信息，进行相关的语义检查和代码的翻译。代码生成阶段，产生的临时变量也需要保存到符号表。

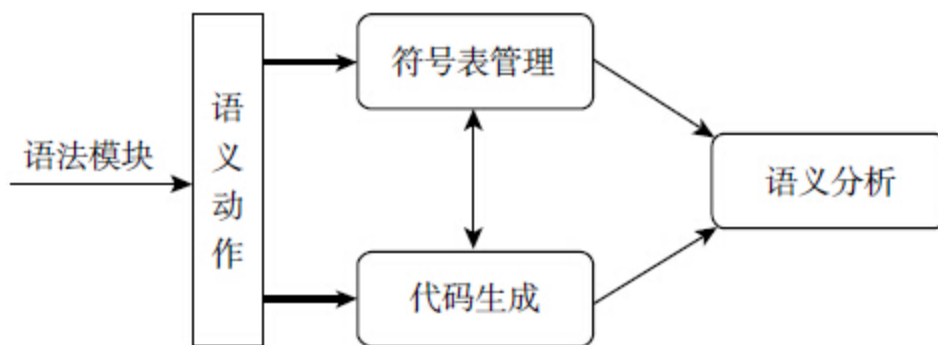


图3-18 语义动作相关模块

3.3.1 符号表数据结构

根据已设计的自定义语言特性，变量、函数和字符串常量的信息是关键的符号信息。另外，由于允许在不同的作用域定义、使用相同的符号名，因此需要在变量符号内保存作用域的信息以区分同名的变量。符号表内最终需要记录的信息包含变量、函数、字符串常量和作用域信息等。使用按名访问的方式有利于符号信息的查询，因此散列表是实现符号表数据结构的较好选择。

如图3-19所示，符号表内保存了三个重要的数据结构：变量表、函数表和串表。变量表使用散列表实现，保存了变量名与同名变量列表的映射，变量列表内保存了变量对象。图中展示了三个名字为`var1`的变量对象，它们的作用域路径分别是`"/0"`、`"/0/1"`、`"/0/2/3"`，类型分别为`"int"`、`"char"`、`"int*"`。函数表也使用散列表实现，保存了函数名与函数对象的映射。图中展示了名为`fun1`的函数对象，它的返回类型是`"int"`，参数列表保存在`args`字段内。串表使用链表实现，保存了程序中定义的字符串常量。图中展示了两个字符串常量`"Hello"`和`"%d"`。

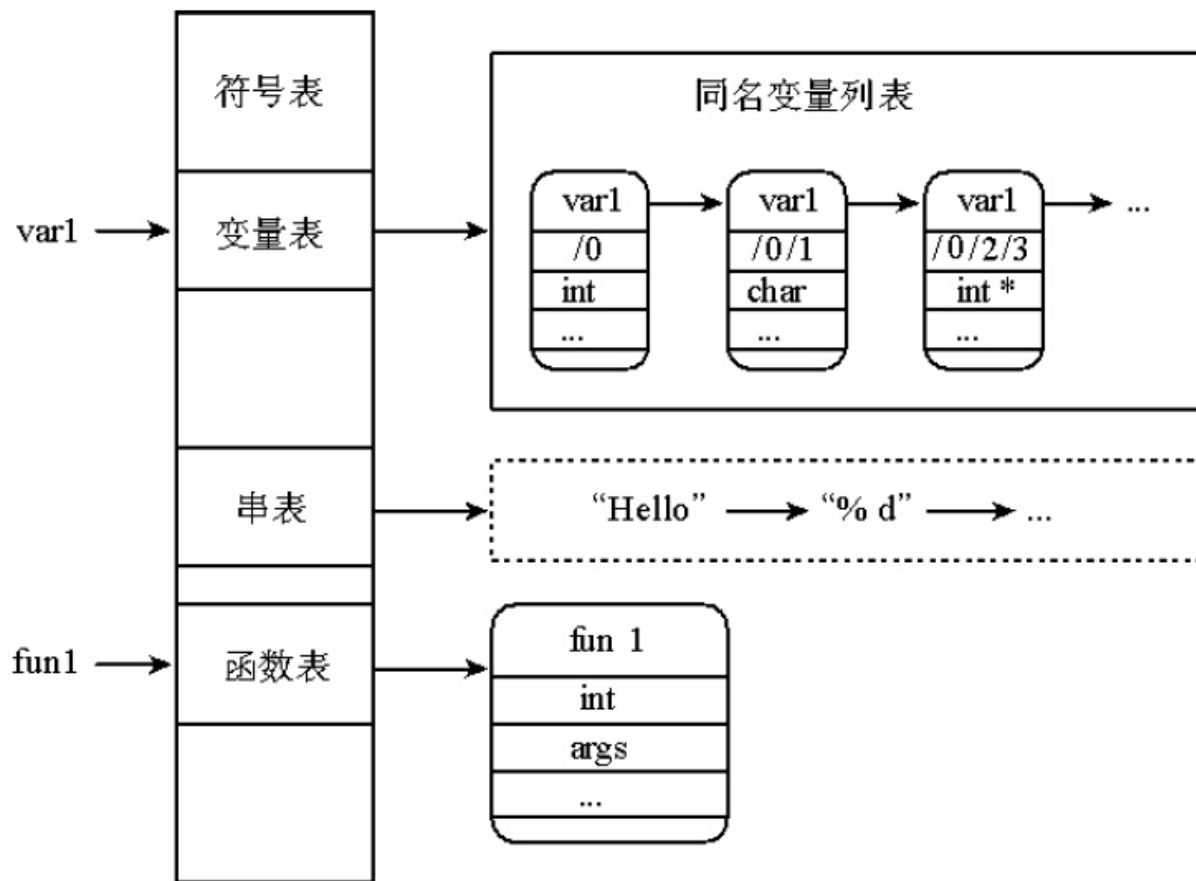


图3-19 符号表结构

符号表相关数据结构的部分实现代码如下:

```

1  /*
2   符号表

3  */
4  class SymTab
5  {
6      struct string_hash{
7          size_t operator()(const string& str) const{
8              return __stl_hash_string(str.c_str());
9          }
10         };
11         hash_map<string,vector<Var*>*,string_hash> varTab;           //变量表
12         hash_map<string,Var*,string_hash> strTab;                  //字符串常量表

```


[illegible]

```

32         union{
//int、
char初值

33             int intVal;
34             char charVal;
35         };
36         string strVal;                                //字
字符串常量初值

37         string ptrVal;                                //字
符指针初值

38         Var*ptr;                                        //变
量的指针变量

39         int size;                                      //变
量的大小

40         int offset;                                    //变
量的栈帧偏移

41     };
42     /*
43     函数

44     */
45     class Fun

46     {
47         bool externed;                                //是
否
extern声明

48         Tag type;                                      //返
回类型

49         string name;                                    //函
数名称

50         vector<Var*>paraVar;                            //形参变量列表

51         int maxDepth;                                    //栈
的最大深度

52         int curEsp;                                      //当
前栈指针位置

53         vector<int>scopeEsp;                            //作用域栈指针

```

位置

```
54         vector<InterInst*> interCode;                                //目标代码

55         InterInst* returnPoint;                                       //返
回点

56     };
```

第1~16行描述了符号表SymTab数据结构内的关键字段。

第5~9行定义了字符串的hash函数对象string_hash，用于将字符串转化为一个无符号整数值。

第10~12行定义了变量表varTab、字符串常量表strTab和函数表funTab。

第13行curFun记录当前分析的函数。

第14~15行中，scopeId记录作用域的唯一编号，scopePath记录从全局作用域到当前作用域的路径。变量表由散列表实现，元素类型为vector，记录了同名变量对应的Var对象指针。根据C语言定义，不同作用域内是允许出现同名变量的，而且内部作用域将覆盖外部作用域的同名变量。因此，必须使用作用域路径区分不同作用域的同名变量。

第17~41行描述了变量数据结构的关键字段。

字段literal表示Var对象是否是常量，递归下降子程序literal识别常量时创建的Var对象的literal字段总为true，其他情况该字段为false。

字段scopePath记录了变量声明或定义时所在的作用域路径，同名变量通过该作用域字段区分自己的作用范围。

第23~25行定义的字段externed、type和name分别表示变量是否是extern声明形式、类型和变量名基本信息。

第26~28行中，字段isPtr表示变量是否是指针类型。字段isArray表示变量是否是数组类型，如果isArray为true，字段arraySize表示数组的大小。

字段isLeft用于表示变量是否可以作为左值，即是否允许被其他表达式赋值或者被取地址等，用于表达式语义的检查。

字段initData记录了变量定义时的初始化表达式的结果变量。

第32~35行的匿名联合记录了int和char类型变量的初值。

字段strVal记录了字符串常量的内容。

字段ptrVal记录了字符指针的初值，即初始化字符串常量的名称。

字段ptr记录了指向当前变量的指针变量。例如指针运算*p的结果变量为t，则变量t的Var对象的ptr字段指向变量p的Var对象。

字段size记录变量的大小，单位为字节。

字段offset记录局部变量或参数变量相对于栈帧基址的偏移量，如果变量是全局变量，该字段为0。

第42~56行描述了函数数据结构的关键字段。

第47~49行中，字段externed、type和name分别表示函数是否使用extern声明、返回类型和函数名。

字段paraVar记录了函数形式参数变量列表。

字段maxDepth记录了函数栈帧的最大值，即需要开辟的栈帧大小。

字段curEsp记录了当前栈指针的位置，即当前栈帧的大小，初值为栈帧基址的位置。

字段scopeEsp动态记录每个作用域的大小。进入作用域i时，设定scopeEsp[i]初值为0。离开作用域时，将curEsp减去scopeEsp[i]恢复到进入作用域之前的栈帧大小。

字段interCode记录生成的目标代码。

字段returnPoint记录函数返回时需要跳转到的函数退出代码位置。

3.3.2 作用域管理

对代码作用域的管理是为了区分不同作用域的同名变量，以及确定局部变量相对于栈帧基址的偏移。

在C语言中，作用域具有以下特点。

- 1) 一般使用花括弧对“{}”表示一个作用域。
- 2) 作用域允许嵌套。
- 3) 作用域内声明的变量在作用域外不可见。
- 4) 对于嵌套作用域，外部作用域声明的变量在内部作用域可见。
- 5) 内部作用域声明的变量可以覆盖外部作用域声明的同名变量。

一般情况下，很少使用花括弧对“{}”直接定义代码作用域，而是使用常用的复合语句，例如**while**循环、**if-else**条件语句等定义代码作用域。

```
while(condition)
{
    statements
}
```

比如while循环语句，虽然condition表达式不在花括弧对内部，但仍属于while循环的作用域，即与statements在一个作用域内。这是因为，对while循环语句来说，其内部只有一个作用域——循环体，循环条件部分可以直接与循环体合并。类似的情况还有if语句、else语句、for循环语句等。不过有一个语句例外，就是do-while循环语句。

```
do
{
    statements
}
while(condition);
```

如果在do-while循环内将statements和condition看作一个作用域内，那么对于condition表达式来说，statements声明的变量便可以在condition内可见，这违反了作用域的基本特性。因此，condition表达式应该属于do-while循环的外层作用域。

符号表提供了两个基本操作用于作用域的管理。

```
1  /*
2   进入局部作用域

3  */
4  void SymTab::enter
5  (
6  {
7      scopeId++;
8      scopePath.push_back(scopeId);
9      if(curFun)curFun->enterScope();
10 }
11 /*
12  离开局部作用域

13 */
14 void SymTab::leave
```

```
()
15 {
16     scopePath.pop_back();
17     if(curFun)curFun->leaveScope();
18 }
```

作用域管理的思想很简单，由于作用域支持嵌套结构，使用栈描述作用域的动态变化最为合适。代码中`scopePath`保存了当前作用域的嵌套结构，其内部元素是每个作用域的编号，我们为每个作用域分配一个唯一的编号，存放在符号表的`scopeId`字段。

使用`enter`函数进入一个新的作用域，并改变`scopeId`，表示当前作用域的编号。`scopeId`初值为0，表示全局作用域。然后将作用域编号放入栈中，这样`scopePath`反映的作用域嵌套结构正好是全局作用域到当前作用域的路径。在当前作用域声明的变量都会保存这个作用域路径，用于区分不同作用域下的同名变量。

使用`leave`函数离开作用域，只需要将`scopePath`栈顶元素出栈即可，这样`scopePath`便恢复到进入这个作用域前的状态，即上级作用域的路径。

在语法分析的递归下降子程序中，通过插入作用域管理的代码完成相应的语义动作。例如`while`循环的递归下降子程序插入作用域管理代码后形式如下：

```
1 void Parser::whilestat
   ()
2 {
```



```

3      symtab.enter

();
4      match(KW_WHILE);
5      if(!match(LPAREN))
6          recovery(EXPR_FIRST||F(RPAREN)
7                  ,LPAREN_LOST,LPAREN_WRONG);
8      altexpr();
9      if(!match(RPAREN))
10         recovery(F(LBRACE),RPAREN_LOST,RPAREN_WRONG);
11  block();
12  symtab.leave

();
13 }

```

这样，在代码第3~12行之间产生的变量都是属于**while**循环作用域的。类似的，**do-while**循环插入作用域管理代码后形式如下：

```

1  void Parser::dowhilestat()
2  {
3      symtab.enter

();
4      match(KW_DO);
5      block();
6      if(!match(KW_WHILE))
7          recovery(F(LPAREN),WHILE_LOST,WHILE_WRONG);
8      if(!match(LPAREN))
9          recovery(EXPR_FIRST||F(RPAREN)
10                 ,LPAREN_LOST,LPAREN_WRONG);
11  symtab.leave

();
12  altexpr();
13  if(!match(RPAREN))
14      recovery(F(SEMICON),RPAREN_LOST,RPAREN_WRONG);
15  if(!match(SEMICON))
16      recovery(TYPE_FIRST||STATEMENT_FIRST||F(RBRACE)
17              ,SEMICON_LOST,SEMICON_WRONG);
18 }

```

可以看出，作用域管理代码仅作用于代码第3~11行之间，第12行后的表达式代码并不在**do-while**的内部作用域内，而是处于上级作用域的范围。

函数定义和声明的作用域管理有一点特别，它与while循环的作用域管理类似。我们为非终结符<idtail>定义的产生式为：

```
<idtail>-><varrdef><deflist> | LPAREN <para> RPAREN <funtail>
```

对应的递归下降子程序插入作用域管理代码后形式为：

```
1 void Parser::idtail()
2 {
3     if(match(LPAREN)){
4         symtab.enter
5
6         ();
7         para();
8         if(!match(RPAREN))
9             recovery(F(LBRACK)_(SEMICON)
10                    , RPAREN_LOST, RPAREN_WRONG);
11         funtail();
12         symtab.leave
13
14         ();
15     }
16     else{
17         varrdef();
18         deflist();
19     }
20 }
```

函数的参数变量属于函数内部作用域，为了将函数的参数变量包含到函数作用域内部，我们将作用域管理代码插入para和funtail前后。这样做的结果是，无论是函数声明还是函数定义都会产生新的作用域，不过这并不影响作用域的管理。

如图3-20所示的代码，为了简便起见，这里只保留了变量的声明信息。按照上述变量作用域的管理方式，示例代码会产生4个代码作用域：全局作用域（编号0）、main函数作用域（编号1）、fun函数作用

域（编号2）和if语句作用域（编号3）。符号表初始化时，将0号作用域入栈，进入main函数时将1号作用域入栈，得到main函数作用域路径“/0/1”。当离开main函数时，1号作用域出栈，作用域路径恢复为“/0”。接着进入fun函数作用域时，作用域路径变为“/0/2”，再进入if语句作用域时，作用域路径变为“/0/2/3”。这样在全局作用域、main函数作用域和fun函数的if语句作用域内定义的同名变量var1拥有不同的作用域路径“/0”、“/0/1”和“/0/2/3”，完成了不同作用域同名变量的区分。

代码与作用域		路径	变量
int var1;	①	/0	var1:/0
<div><div>int main(){ char var1; }</div><div>int fun(){ if(x){ int* var1; } }</div></div>	①	/0/1	var1:/0/1
		/0	
	②	/0/2	
	③	/0/2/3	var1:/0/2/3
		/0/2	
		/0	

图3-20 变量作用域路径

前面讲到，作用域管理除了区分同名变量，还具有计算局部变量相对于栈帧基址的偏移的作用。而在符号表的enter和leave函数内分别

调用了Fun对象的enterScope和leaveScope的函数正是完成了局部变量栈帧内偏移的计算。

```
1  /*
2      进入一个新的作用域

3  */
4  void Fun::enterScope

5  ()
6  {
7      scopeEsp.push_back(0);
8  }
9  /*
10  离开当前作用域

11 */
12 void Fun::leaveScope

13 ()
14 {
15     maxDepth=(curEsp>maxDepth)?curEsp:maxDepth;
16     curEsp-=scopeEsp.back();
17     scopeEsp.pop_back();
18 }
```

与使用scopePath记录作用域路径类似，scopeEsp动态保存了每个作用域的大小，且按照作用域的嵌套层次进行管理。

每次进入新的作用域时，作用域的大小为0，scopeEsp将0入栈。此后凡是在作用域内定义了新的变量，则将该变量大小累加到当前作用域的大小，同时curEsp也要累加变量的大小。这样scopeEsp内总是保存着当前所有作用域的实际大小，而curEsp一直反映栈帧的深度。

每次离开作用域时，首先计算栈帧的最大深度，然后将curEsp减去当前作用域的大小（栈顶元素），恢复到进入作用域前栈帧深度，

最后将当前作用域的大小从栈内弹出。

如图3-21所示，继续前面的代码示例。变量的栈帧内偏移仅局限于函数内部，因此全局变量不需要考虑这个问题。在main函数内，作用域大小为4，栈帧最大深度为4。进入fun函数作用域时，scopeEsp内压入0，然后进入if语句作用域，继续压入0，scopeEsp内保存的作用域大小为“0-0”。当定义变量var1后，if语句作用域大小增加，scopeEsp变为“0-4”，curEsp增为4。当离开if语句作用域后，curEsp减去if语句作用域大小4，变为0，同时scopeEsp弹出栈顶元素，恢复为“0”。这样，curEsp总是保存局部变量相对于栈帧基址的偏移。

代码与作用域		scopeEsp	curEsp	maxEsp
int var1;	①			
int main(){ char var1; }	①	0	0	4
		4	4	
int fun(){	②	0	0	4
if(x){ int* var1; }	③	0-0	0	
		0-4	4	
}		0	0	

图3-21 变量栈帧内偏移

3.3.3 变量管理

变量管理涉及变量对象的创建、将变量对象添加到变量表varTab和从变量表varTab取出变量对象。不过这些操作并不是简单地对数据结构的增、删、改、查，变量对象创建时需要考虑初始化的不同情况，增加变量对象时需要检查变量的声明合法性，获取变量对象时需要检查变量引用的合法性等。

1.创建变量对象

变量对象的来源有三种，编译器对不同来源的变量处理不同。

- 1) 在源程序内显式声明变量。比如“`extern int var;`”或“`int x=1;`”等，这类变量的变量名是程序内显式指定的标识符ID，包括全局变量、参数变量和局部变量。
- 2) 源程序内定义的常量。作为表达式运算的基本单位，常量被看作特殊的变量，这类变量只有值而没有显式的名字，编译器需要为之指定一个名字。
- 3) 表达式运算的临时结果变量。比如表达式“`a+b+c`”，按照加法运算符的结合性，子表达式“`a+b`”优先计算，结果需要保存到临时变量，临时变量也没有显式的名称，需要编译器指定唯一的名字。

在自定义语言文法定义中，使用非终结符<defdata>表示一个完整的全局变量或局部变量的形式，包括声明、定义和初始化。使用非终结符组合“<type><paradata>”表示一个完整的形式参数变量的形式。

非终结符<defdata>的递归下降子程序的实现如下：

```
1  Var* Parser::defdata
   (bool ext, Tag t){
2      string name="";
3      if(F(ID)){
4          name=((Id*)look)->name);
5          move();
6          return varrdef
   (ext, t, false, name);
7      }
8      else if(match(MUL)){
9          if(F(ID)){
10             name=((Id*)look)->name);
11             move();
12         }
13         else
14             recovery(F(SEMICON)_(COMMA)_(ASSIGN)
15                     , ID_LOST, ID_WRONG);
16         return init
   (ext, t, true, name);
17     }
18     else{
19         recovery(F(SEMICON)_(COMMA)_(ASSIGN)_(LBRACK)
20                 , ID_LOST, ID_WRONG);
21         return varrdef
   (ext, t, false, name);
22     }
23 }
24 Var* Parser::varrdef
   (bool ext, Tag t, bool ptr, string name){
25     if(match(LBRACK)){
26         int len=0;
27         if(F(NUM)){
28             len=((Num*)look)->val;
29             move();
30         }
31         else
32             recovery(F(RBRACK), NUM_LOST, NUM_WRONG);
33         if(!match(RBRACK))
34             recovery(F(COMMA)_(SEMICON)
35                     , RBRACK_LOST, RBRACK_WRONG);
36         return new Var(symtab.getScopePath()
37
38             , ext, t, name, len);
```

```

//新的数组

38     }
39     else
40         return init

(ext,t,ptr,name);
41 }
42 Var* Parser::init

(bool ext,Tag t,bool ptr,string name){
43     Var* initVal=NULL;
44     if(match(ASSIGN)){
45         initVal=expr();
46     }
47     return new Var(symtab.getScopePath()

48         ,ext,t,ptr,name,initVal);

//新的变量或者指针

49 }

```

回顾前面对<defdata>文法的定义。

```

<defdata>->ID <varrdef> | MUL ID <init>
<varrdef>->LBRACK NUM RBRACK | <init>
<init>->ASSIGN <expr> | ε

```

虽然<defdata>表达了所有显式声明变量的形式，但是非终结符<varrdef>和<init>才最终确定了变量的形式。前者用于确定是变量还是数组，后者确定变量或指针的初始化部分（我们定义的文法不允许数组的初始化）。

子程序defdata的参数ext用来表示被分析的变量是否由extern声明，参数t用来表示变量的类型。这两个参数由调用defdata的子程序传递。而defdata则将这些变量信息传递给子程序varrdef和init，当它们获

得了变量的所有信息后，就创建新的变量对象Var，并将收集到的变量信息复制到Var对象的对应字段内。

对于常量，创建变量对象的方式较为简单。

```
1  Var::Var(Token*lt){
2      clear();
3      literal=true;                                //常量标记

4      setLeft(false);                             //不能作为左值

5      switch(lt->tag){
6          case NUM:

7              setType(KW_INT);
8              name("<int>");                        //类型作为名字

9              intVal=((Num*)lt)->val;               //记录数字数值

10             break;
11         case CH:

12             setType(KW_CHAR);
13             name("<char>");                        //类型作为名字

14             intVal=0;                              //高位置
0
15             charVal=((Char*)lt)->ch;              //记录字符值

16             break;
17         case STR:

18             setType(KW_CHAR);
19             name=GenCode::genLb();                 //产生一个新的名字

20             strVal=((Str*)lt)->str;                //记录字符串值

21             setArray(strVal.size()+1);             //字符串作为字符数组存储

22             break;
23     }
24 }
```

在词法分析器中，常量的所有信息都保存在Token对象内，因此可以直接从Token对象创建对应的变量对象。我们为所有的整数或字符常量设定了同一个名字“<int>”或“<char>”，这样在符号表数据结构的变量表内，它们被保存在键值为“<int>”或“<char>”的同名链表内。因为不会存在对常量的按名访问，这样做无可厚非。但是对于字符串常量而言，就需要为每个字符串分配一个唯一的名字，这是因为后面代码生成时需要根据这个名字确定字符串的起始地址。代码生成器中，使用genLb函数为字符串生成一个唯一的名字。

存放表达式结果的临时变量与表达式的代码生成关系很大，在后面会详细描述。

2.添加变量对象

经过defdata、paradata、literal等子程序创建了变量对象后，符号表类SymTab提供的addVar函数将新创建的变量对象添加到变量表varTab中。

```
1 void SymTab::addVar
  (Var* var){
2     if(varTab.find(var->getName())==varTab.end()){
3         varTab[var->getName()]=new vector<Var*>;           //创建链表

4         varTab[var->getName()]->push_back(var);           //添加变量

5     }
6     else{
7         vector<Var*>&list=*varTab[var->getName()];         //同名变量列表
```

```

8          int i;
9          for(i=0;i<list.size();i++)                //判断变量
作用域

10              if(list[i]->getPath().back()==var->getPath().back())
11                  break;
12          if(i==list.size()||var->getName()[0]=='<')    //排除常量

13              list.push_back(var);
14          else{
15              SEMERROR(VAR_RE_DEF, var->getName());    //变量重定
义

16              delete var;
17              return;
18          }
19      }
20      if(ir){
21          int flag=ir->genVarInit
(var);                //变量初始化语句

22          if(curFun&&flag)curFun->locate
(var);                //计算局部变量栈帧偏移

23      }
24  }

```

第2~5行处理变量表**varTab**内不存在添加变量**var**名称的同名变量列表时的情况。首先创建同名变量列表，添加到变量表，然后将**var**添加到同名变量列表内。

第6~19行处理同名变量列表存在时的情况，此时需要判断变量作用域的合法性，即不允许同一个作用域下出现同名的变量。

第7行获取同名变量列表**list**。

第9~11行遍历**list**，并将**list**内保存的变量对象的作用域路径与**var**的作用域路径进行匹配。我们知道，作用域路径表示全局作用域到当

前作用域的路径，而每个作用域分配了唯一的编号，因此通过比较作用域最后一个元素便可以确定作用域路径是否相同。循环退出时，如果索引*i*不等于同名列表的长度，则表示出现了相同作用域的同名变量，需要报告语义错误。

第12~13行表示不存在同作用域的同名变量，将变量添加到同名变量列表。前面提到整数和字符常量都保存在“<int>”和“<char>”同名变量列表内，且作用域都为空，因此会导致索引*i*不等于同名列表的长度，触发语义错误。为了避免这一点，我们添加了对变量名的判断，即判定名字的第一个字符是否是'<'，因为标识符名称是不可能以'<'开始的。

第14~17行处理变量重定义语义错误，语义错误的内容在后面会详细描述。

第21行处理变量的初始化语句。自定义语言语法规则规定全局变量的初始值只能是常量，而不能是表达式。而局部变量可以使用任意合法的表达式进行初始化，虽然它们在文法形式上完全相同（都是由defdata定义）。在Var数据结构内，initData保存了表达式的结果变量，如果该变量是常量值，则直接将值复制到被初始化的变量对象内，否则需要生成赋值语句完成初始化操作，在后面阐述代码生成时会对此进行详细描述。

第22行调用locate处理局部变量的栈帧偏移地址，代码如下：

```
1 void Fun::locate
(Var*var){
2     int size=var->getSize();
3     size+=(4-size%4)%4; //按照
4字节的大小整数倍分配局部变量

4     scopeEsp.back()+=size; //累加作用域大小

5     curEsp+=size; //累加栈指针位置

6     var->setOffset(-curEsp); //局部变量偏移为负数

7 }
```

函数**locate**首先获取变量的大小，保存到**size**中，并将**size**按照4字节对齐。然后修改当前作用域的大小和栈指针的位置，最后将栈指针位置的负值保存到局部变量的栈帧偏移字段**offset**内。之所以保存负值，是因为栈是从高字节到低字节增长的，局部变量的栈帧偏移从0开始分配。关于函数栈帧机制，在代码生成章节会详细描述。

在将常量添加到符号表时，需要考虑符号表中字符串常量的行为：

```
1 void SymTab::addStr
(Var* &v){
2     hash_map<string,Var*,string_hash>::iterator strIt,
3         strEnd=strTab.end();
4     for(strIt=strTab.begin();strIt!=strEnd;++strIt){
5         Var*str=strIt->second;
6         if(v->getStrVal()==str->getStrVal()){
7             delete v;
8             v=str;
9         }
10    }
```

常量已存在 //字符串

```

9             return;
10        }
11    }
12    strTab[v->getName()]=v;
13 }
14 Var* Parser::literal

(){
15     Var *v =NULL;
16     if(F(NUM)_(STR)_(CH)){
17         v = new Var(look);
18         if(F(STR))
19             symtab.addStr(v);                //字符
常量记录

20     else
21         symtab.addVar(v);                    //其他
常量也记录到符号表

22     move();
23 }
24 else
25     recovery(RVAL_OPR, LITERAL_LOST, LITERAL_WRONG);
26     return v;
27 }

```

子程序**literal**识别所有的常量，第17行根据常量的**Token**对象创建变量对象**v**。将字符串常量添加到字符串常量表**strTab**中，而将其他常量正常添加到变量表**varTab**中。

函数**addStr**用于添加一个字符串常量对象。第4~5行遍历字符串常量表，将取出的每个字符串常量存入**str**中。第6行判断待添加的字符串常量**v**是否已经存在于字符串常量表**strTab**中，如果存在则删除**v**，并将**v**设置为已经存在的字符串常量**str**。如果**strTab**内不存在字符串常量**v**，则将**v**添加到**strTab**中，键值为**v->name**。

3.获取变量对象

由于允许不同作用域的同名变量覆盖，因此获取一个变量对象需要提供两个基本信息：变量名和变量访问作用域。变量表使用散列表和同名变量列表组合的方式组织，通过变量名确定散列表内的同名变量列表，再根据变量访问作用域确定具体的变量对象。符号表提供 `getVar` 函数用于获取变量对象。

```
1  Var* SymTab::getVar
   (string name){
2      Var*select=NULL;                                //最佳选择

3      if(varTab.find(name)!=varTab.end()){
4          vector<Var*>&list=*varTab[name];
5          int pathLen=scopePath.size();                //当前路径
   长度

6          int maxLen=0;                                //已经匹配
   的最大长度

7          for(int i=0;i<list.size();i++){
8              int len=list[i]->getPath().size();
9              if(len<=pathLen&&
10                 list[i]->getPath()[len-1]==scopePath[len-1]){
11                 if(len>maxLen){                       //选取最长匹配

12                     maxLen=len;
13                     select=list[i];
14                 }
15             }
16         }
17     }
18     if(!select)
19         SEMERROR(VAR_UN_DEC,name);                   //变量未声明

20     return select;
21 }
```

程序中使用指针变量 `select` 记录最终获取的变量对象，初始化为 `NULL`。

第4行根据变量名访问散列表，获取同名变量列表list。

第5~16行根据当前的作用域路径scopePath获取“最近”的变量对象。通过遍历同名变量列表，选择一个变量，该变量的作用域路径与当前作用域路径scopePath匹配度最高。

举个例子来说，假设获取名为“var”的变量对象，在var的同名变量列表内保存了四个变量，其作用域路径分别为“/0”、“/0/1”、“/0/2/3/4”、“/0/2”，当前作用域路径为“/0/2/5”。按照最长匹配原则，我们应该选择作用域路径为“/0/2”的变量。首先可以确定待选择的变量作用域路径长度一定不大于当前作用域路径长度，比如路径“/0/2/3/4”的变量在作用域3内，与作用域5是并列的，变量不可见。然后考虑待选择的变量作用域路径一定是当前作用域路径的前缀，否则变量依然不可见，比如作用域“/0/1”。对于作用域路径“/0”，第一个元素与“/0/2/5”的第一个元素相同，因此路径匹配，匹配长度为1，这是因为到达同一个作用域的作用域子路径一定相同。而对于作用域路径“/0/2”，第二个元素与“/0/2/5”的第二个元素相同，因此路径匹配，匹配长度为2。故而，最终选择的变量的作用域路径为“/0/2”。

第9行判断条件选择变量作用域长度不大于当前作用域路径长度的变量。

第10行判断变量作用域路径的最后一个元素是否与当前作用域路径对应的元素相同。

第11~14行选择最长匹配，将变量结果保存到`select`中。

第18行判断是否获取到变量，否则报告变量未声明语义错误。

函数`getVar`的调用时机发生在变量名被引用的表达式中，递归下降子程序`idexpr`中处理了变量和数组的访问，此处便需要根据标识符的名称获取已保存的变量对象。

3.3.4 函数管理

函数管理涉及函数对象的创建、将函数对象添加到函数表`funTab`和从函数表`funTab`取出函数对象。相比而言，函数对象的创建比变量对象的创建简单。而函数对象的添加则较为复杂，这是因为需要考虑函数定义和函数声明的不同。获取函数对象时，除了提供函数名外，还需要提供实际参数列表，以方便符号表对函数参数类型进行检查。

1.创建函数对象

无论是函数定义还是函数声明，它们在文法级别具有公共的首部。在递归下降子程序`idtail`内，完全可以根据函数的首部确定函数的基本要素：函数名、返回值类型和参数列表。

```

1 void Parser::idtail
    (bool ext, Tag t, bool ptr, string name){
2     if(match(LPAREN)){ //函
数
3         symtab.enter();
4         vector<Var*>paraList; //参数列表
5
6         para(paraList);
7         if(!match(RPAREN))
8             recovery(F(LBRACK)_(SEMICON)
9                 , RPAREN_LOST, RPAREN_WRONG);
10        Fun* fun=new Fun(ext, t, name, paraList);
11
12        funtail(fun);
13        symtab.leave();
14    }
15    else{ //变
量

```

```

14             symtab.addVar(varrdef(ext,t,false,name));
15             deflist(ext,t);
16         }
17     }
18 void Parser::funtail

(Fun*f){
19     if(match(SEMICON)){                                     //函
数声明

20             symtab.decFun

(f);
21     }
22     else{
23         symtab.defFun

(f);                //函数定义

24         block();
25         symtab.endDefFun

();                //结束函数定义

26     }
27 }

```

第9行创建了函数对象，传入的参数ext、t、name、paralist分别表示函数是否声明为extern形式、函数的返回类型、函数名和形式参数列表。至于具体fun函数对象是函数声明还是定义，需要由funtail子程序确定。

第20行调用的decFun函数用于将函数对象以声明形式插入函数表。

第23行调用的defFun函数用于将函数对象以定义形式插入函数表。

第25行处理函数定义结束的工作。

创建函数对象时，除了保存函数的基本信息外，还需要为参数变量计算栈帧偏移，以保证函数能正常访问参数变量。

```
1  Fun::Fun
   (bool ext, Tag t, string n, vector<Var*>&paraList)
2  {
3      externed=ext;
4      type=t;
5      name=n;
6      paraVar=paraList;
7      curEsp=0;
8      maxDepth=0;
9      for(int i=0, argOff=8; i<paraVar.size(); i++, argOff+=4){
10         paraVar[i]->setOffset(argOff);
11     }
12 }
```

第3~6行保存了函数的基本信息，第7~8行将curEsp和maxDepth初始化为0。

第9~10行计算参数相对于栈帧基址的偏移，参数传递都是固定的4字节大小，且栈帧偏移都是正值，从8字节开始。在代码生成中会详细描述函数栈帧。

2.添加函数对象

首先看声明函数对象的添加，decFun实现代码如下：

```
1  void SymTab::decFun
   (Fun* fun){
2      fun->setExtern(true);
3      if(funTab.find(fun->getName())==funTab.end()){           // 未找到函数

4                              funTab[fun->getName()]=fun;       // 添加函数
```

```

5         }
6     else{
7         Fun* last=funTab[fun->getName()];           //获取已经保存的
函数对象

8         if(!last->match
(fun)){
9             SEMERROR(FUN_DEC_ERR, fun->getName());    //函数声明冲突

10        }
11        delete fun;
12    }
13 }

```

首先，第2行将函数对象**fun**的**externed**字段设为**true**，表示一个函数声明。

第3~5行在函数表**funTab**中查找同名的函数对象，如果不存在，则表示第一次声明函数，将该函数插入**funTab**中。

第6~12行处理函数表**funTab**时，若**funTab**中出现了与欲添加函数同名的函数对象，则需要判断当前函数声明是否与原保存的函数对象匹配。

第7行取出已保存的函数对象**last**，第8行使用**match**函数匹配欲添加函数与已保存的同名函数对象的形式，如果匹配失败，则报告函数声明语义错误。

函数**match**的实现如下：

```

1  bool Fun::match
(Fun*f){
2      if(name!=f->name)

```

```

3         return false;
4     if(paraVar.size()!=f->paraVar.size())
5         return false;
6     int len=paraVar.size();
7     for(int i=0;i<len;i++){
8         if(GenIR::typeCheck
(paraVar[i],f->paraVar[i])){           //类型兼容

9             if(paraVar[i]->getType()!=f->paraVar[i]->getType()){
10                 SEMWARN(FUN_DEC_CONFLICT,name);           //函数
声明冲突

11             }
12         }
13         else
14             return false;
15     }
16     if(type!=f->type){           //匹配成功
后再验证返回类型

17         SEMWARN(FUN_RET_CONFLICT,name);           //函数
返回值冲突

18     }
19     return true;
20 }

```

第2~5行验证两个函数的函数名和参数的数量是否相同。

第7~15行验证两个函数的参数类型是否匹配，第8行使用代码生成器的typeCheck函数检查两个函数的类型是否可以相互转换（即兼容，比如类型int*和int[]可以兼容）。第9行表示如果两个类型可以相互转换但是不相同，使用SEMWARN报告“函数声明冲突”语义警告。如果参数列表内有一个参数类型不能兼容，则表示函数声明不能正确匹配。

第16~18行检查两个函数返回值的类型，返回类型不能决定函数的唯一形式，当两个函数的返回类型相同时，报告“返回值类型冲

突”语义警告。

定义函数对象的添加分为两个部分来完成：defFun和endDefFun。

```
1 void SymTab::defFun
  (Fun* fun){
2     if(fun->getExtern()){                                //extern
    不允许出现在定义

3         SEMERROR(EXTERN_FUN_DEF, fun->getName());
4         fun->setExtern(false);
5     }
6     if(funTab.find(fun->getName())==funTab.end()){
7         funTab[fun->getName()]=fun;                    //添加函数

8         funList.push_back(fun->getName());
9     }
10    else{                                                //已经声
    明

11        Fun*last=funTab[fun->getName()];
12        if(last->getExtern()){                            //之前是声明

13            if(!last->match
    (fun)){        //不匹配声明

14                SEMERROR(FUN_DEC_ERR, fun->getName());
15            }
16            last->define
    (fun);        //保存定义信息

17        }
18    } else{
    //重定义

19        SEMERROR(FUN_RE_DEF, fun->getName());
20    }
21    delete fun;
    //删除当前函数对象

22    fun=last;
    //公用函数结构体

23    }
24    curFun=fun;
    //当前分析的函数
```

```

25         ir->genFunHead
(curFun);           //产生函数入口

26 }
27 void Fun::define
(Fun*def){
28     externed=false;
//定义

29     paraVar=def->paraVar;           //
拷贝参数

30 }
31 void SymTab::endDefFun
(){
32     ir->genFunTail
(curFun);           //产生函数出口

33     curFun=NULL;
//当前分析的函数置空

34 }

```

函数defFun将定义函数对象插入函数表。第2~5行检查函数首部是否包含了extern关键字，如果包含则报告语义错误，并将函数对象的externed字段设为false，表示该函数是被定义的。在文法定义章节中，我们讨论了带extern的函数定义形式，针对externed字段的语义检查便是对语法分析的补充。

第6~9行表示当前名称的函数对象首次添加到函数表。

第10~20行处理函数表中已有同名函数对象的情况。如果函数表内存在的函数对象是函数声明，则当前添加的函数定义是合理操作，只需要检查函数形式是否匹配即可。如果函数表内存在的函数对象是

函数定义，则当前添加函数定义是非法操作。C语言不提供函数重载的机制，因此会报告函数重定义语义错误。

第16行调用**define**函数将函数定义的信息保存到原有的函数对象。**define**函数的实现在第27~30行，即设置**externed**字段为**false**，并将函数定义的参数列表保存到函数表内存储的函数对象内。这是因为函数体内的代码要用到参数的名称，原有的函数声明中参数名字已经无效。

第25行调用代码生成器的**genFunHead**产生函数的首部，在代码生成章节中会详细描述。

函数**endDefFun**处理函数定义结束后的工作。首先为当前函数对象**curFun**产生函数的尾部，然后将**curFun**指针置为**NULL**，表示当前作用域离开了函数作用域，并进入了全局作用域。

3. 获取函数对象

由于函数对象是直接插入函数表中的，因此使用函数名可以唯一确定函数对象。在语法分析中，访问函数对象的时机是在函数调用的时候，因此获取函数对象时还需要额外检查函数调用的实际参数类型是否与函数声明的形式参数类型匹配。

```
1  Fun* SymTab::getFun
   (string name,vector<Var*>& args){
2      if(funTab.find(name)!=funTab.end()){
3          Fun* last=funTab[name];
```

```

4             if(!last->match
(args)){
5                 SEMERROR(FUN_CALL_ERR, name);           //形参与实参不匹
配

6                 return NULL;
7             }
8             return last;
9         }
10        SEMERROR(FUN_UN_DEC, name);                       //函数未声明

11        return NULL;
12    }

```

第2行中根据函数名`name`取出函数表`funTab`的函数对象`last`，如果函数对象不存在则报告“函数未声明”语义错误。

第4行检查实际参数列表是否与形式参数列表匹配，如果不匹配报告语义错误。

```

1    bool Fun::match
(vector<Var*>&args){
2        if(paraVar.size()!=args.size())
3            return false;
4        int len=paraVar.size();
5        for(int i=0;i<len;i++){
6            if(!GenIR::typeCheck
(paraVar[i],args[i]))           //类型检查不兼容

7                return false;
8        }
9        return true;
10    }

```

第2~3行检查实参和形参列表的长度是否相同。

第5~8行遍历参数列表，并使用`typeCheck`函数检测每个形式参数和实际参数类型是否兼容。

3.4 语义分析

在符号表管理中，涉及了很多语义错误的处理。根据图3-18描述的语义动作，语义分析是“穿插”在符号表管理和代码生成中的。比如在获取函数对象时，会检查函数对象是否存在，这本身就是语义分析的流程。本节所述的语义分析是解决语法分析不能或者很难处理的上下文相关信息，而语义分析的实现则由具体的符号表管理功能和代码生成功能来完成。

客观地说，语义分析是符号表管理和代码生成过程中，对不满足既定语言特性的处理。比如，根据文法定义，代码“`void x;`”是合法的。在创建变量对象时需要记录变量的类型`void`，此时符号表需要判断类型的合法性，指出`void`类型的变量不合法。换句话说，如果代码中不存在语义错误，符号表完全可以无视`void`类型变量的情况，直接记录变量的类型。显然这并不现实，符号表管理和代码生成必须检查输入代码的合法性，就像处理程序中潜在的bug那样，保证后继工作的顺利进行，这就是语义分析。

在我们实现的编译器中，从三个方面检查程序语义的合法性：声明与定义、表达式和语句。

3.4.1 声明与定义语义检查

在符号表管理中，我们列举的代码中涉及了若干声明与定义类语义错误。包括变量重定义、函数重定义、变量未声明、函数未声明、函数声明与定义不匹配、函数定义不允许使用`extern`等。此外，声明与定义的语义检查还包括变量不能是`void`类型、数组长度必须是正整数、变量声明时不允许初始化、全局变量初始值不是常量、变量初始化类型错误。

变量对象构造时，会调用`setType`函数处理类型信息。

```
1 void Var::setType
(Tag t){
2     type=t;
3     if(type==KW_VOID){                                //void变量

4         SEMERROR(VOID_VAR,
5             "");    //不允许使用
6         void变量

7         type=KW_INT;                                    //默认为
8         int
9         }
10        if(!externed&&type==KW_INT)size=4;                //整数
11        4字节

12        else if(!externed&&type==KW_CHAR)size=1;          //字符
13        1字节

14    }
```

第2行记录了变量类型type，第3~6行处理void类型变量的语义错误，并将void变量转化为int类型，第7~8行计算变量的大小。

对于数组变量，需要在声明时指定数组长度（长度为常量）。

```
1 void Var::setArray
  (int len){
2     if(len<=0){
3         SEMERROR(ARRAY_LEN_INVALID
, name); //数组长度小于等于
0错误

4         return ;
5     }
6     else{
7         isArray=true;
8         isLeft=false; //数组不能
          作为左值

9         arraySize=len;
10        if(!externed)size*=len; //类型大
          小乘以数组长度

11    }
12 }
```

第2~5行判断数组长度不大于0时，报告语义错误。

第6~11行记录数组的基本信息，其中isLeft=false表示数组不能作为左值，计算数组大小时则是将数组类型大小与数组长度相乘。

变量声明时初始化部分由setInit处理。

```
1 bool Var::setInit
  (){
2     Var*init=initData; //取
          出初值数据
```

```

3         if(!init)return false;
4         init=false;
5         if(externed)
6             SEMERROR(DEC_INIT_DENY,
name);                                     //声明不允许初始化

7         else if(!GenIR::typeCheck
(this,init))
8             SEMERROR(VAR_INIT_ERR,
name);                                     //类型检查不兼容

9         else if(init->literal){                                     //初值为
常量

10             init=true;
11             if(init->isArray)                                     //初值
是数组，必是字符串

12                 ptrVal=init->name;                                     //字
符指针初值

=常量字符串名

13             else

14                 intVal=init->intVal;                                     //
复制数值数据

15         }
16         else{
// 初值不是常量

17             if(scopePath.size()==1)                                     //初始化
全局变量

18                 SEMERROR(GLB_INIT_ERR,
name);                                     //全局变量初始化必须是常量

19             else
//初始化局部变量

20                 return true;
21         }
22         return false;
23     }

```

语法分析的init子程序将变量的初值数据记录到initData中，setInit将initData指向的变量对象数据取出，进行变量初始化工作。

第5~6行表示变量对象的externed字段为true，为变量声明，不能进行初始化。

第7~8行检查初值变量对象类型是否与被初始化的变量对象类型兼容。

第9~15行考虑初值变量对象是常量的情况。

第11~12行表示初值变量既是常量又是数组，满足这两个条件的只有字符串常量。由于文法定义中未涉及数组变量初始化的语法，而前面的类型检查已经通过，因此使用常量字符串初始化的变量一定是字符指针。init的name字段便是常量字符串的名称，使用ptrVal字段来记录初值字符串的名称。

第13~14行表示初值变量是基本类型的常量，因此直接复制初值数据intVal即可。

第16~21行处理初值不是常量的情况。

第17~18行表示被初始化的是全局变量，由于自定义语言规定全局变量不能使用非常量初始化，因此这里报告语义错误。

第19~20行处理局部变量的初始化，由于局部变量的初值不是常量，即局部变量的初始值是一个表达式。因此需要先计算初值表达式的值，然后使用赋值语句对局部变量初始化。此处返回true，表示setInit调用结束后，需要生成initData到this的赋值语句。

在符号表管理的添加变量对象章节描述addVar函数实现时，提到的函数genVarInit是setInit函数的调用者。

```
1  bool GenIR::genVarInit
   (Var*var){
2      if(var->getName()[0]=='<')return false;
3      symtab.addInst(new InterInst(OP_DEC,var));
4      if(var->setInit())                                //初
   初始化

5          genTwoOp

   (var,ASSIGN
6          ,var->getInitData());                        //赋
   值语句

   name=init->name
7       return true;
8  }
```

在函数genVarInit中，首先处理整型常量和字符常量的变量对象，它们不需要初始化。

第3行生成了变量的定义指令，以方便代码生成时处理变量的初始化。

第4~6行调用setInit对变量对象初始化，如果返回值为true，则表示使用了非常量初始化局部变量，因此需要生成赋值语句代码。被赋

值的变量为`var`，值为`var`对象内`initData`指向的变量。函数`genTwoOp`对双操作数运算表达式进行代码生成，代码生成章节会对此进行详细描述。

3.4.2 表达式语义检查

表达式的语义检查包括函数调用时实参与形参的类型不匹配、表达式不能作为左值、函数的void返回值不能参与表达式运算、赋值类型不匹配、表达式不能是基本类型、表达式不是基本类型、数组索引运算错误。在符号表管理中，已经描述了函数实参和形参类型匹配语义的分析。

在赋值表达式中，被赋值的可能是变量（a=1），也可能是数组索引表达式（a[0]=1）或指针表达式（*p=1）。因此，赋值运算符两侧都是表达式形式，文法上我们也没有做额外的限制。但是语义分析时，必须要求被赋值的表达式是左值表达式。除了赋值表达式中需要处理左值表达式，前缀自加自减（++i）、后缀自加自减（i++）和取址运算（&a）也要求运算对象是左值。

```
1  Var* GenIR::genOneOpRight
   (Var*val, Tag opt){
2      if(!val)return NULL;
3      if(val->isVoid
   ()) {
4          SEMERROR(EXPR_IS_VOID
   );                                //void变量
5          return NULL;
6      }
7      if(!val->getLeft
   ()) {
8          SEMERROR(EXPR_NOT_LEFT_VAL
```

```
);
9          return val;
10      }
11      if(opt==INC)return genIncR
(val);                                     //右自加语句

12      if(opt==DEC)return genDecR
(val);                                     //右自减语句

13      return val;
14 }
```

函数genOneOpRight处理单目后缀运算符表达式，即后缀自加自减表达式。

第7~10行调用变量对象的getLeft函数获取判断变量是否是左值，如果不是左值，则报告语义错误。

函数genIncR和genDecR分别处理后缀自加自减表达式的翻译。

在上述代码的第3~6行，判断了变量是否为void类型。void类型的变量并非来源于使用void声明的变量（setType函数将void变量转化为int类型），而是来源于返回值类型为void的函数调用，代码生成中会描述这一实现细节。

赋值表达式运算中，需要进行必要的类型转换，因此赋值运算符两侧的表达式类型必须是兼容的，使用代码生成器的typeCheck函数检查赋值表达式的类型。

指针表达式要求运算的表达式必须是指针或者数组，而不能是基本类型。而在一般的算术表达式中，要求运算的表达式必须是基本类型而非指针或者数组。

数组索引表达式中，对类型的要求比较复杂。

```
1  Var* GenIR::genArray
   (Var*array,Var*index){
2      if(!array || !index)return NULL;
3      if(array->isVoid()||index->isVoid()){
4          SEMERROR(EXPR_IS_VOID
   );
5          return NULL;
6      }
7      if(array->isBase() || !index->isBase()){
8          SEMERROR(ARR_TYPE_ERR
   );
9          return index;
10     }
11     return genPtr
   (genAdd
   (array,index));
12 }
```

在第7~10行，进行对数组索引运算的语义检查。要求数组变量对象array不能是基本类型，索引表达式index必须是基本类型，否则报告语义错误。

第11行生成数组索引表达式的代码，此处将数组索引运算拆分为两步：根据数组名和索引做加法运算得到数组元素地址，再对地址做指针运算获取数组元素的值。

表达式的语义检查和具体的表达式翻译相关性很大，因此在后面表达式的代码生成时，需要留意不同表达式语义检查的内容。

3.4.3 语句语义检查

自定义语言设计的语句基本都是独立的单位，很少存在上下文相关的信息。不过有三个语句较为特殊：`break`、`continue`和`return`语句，它们的出现和位置都有一定的限制，相对应的语义错误是`break`语句不在循环和`switch`语句内、`continue`语句不在循环语句内和`return`语句与函数返回值不匹配。

`break`语句只能出现在循环语句和`switch`语句内部，由于复合语句支持嵌套，因此需要使用与作用域管理类似的机制记录复合语句的嵌套层次。在代码生成中，代码生成器使用栈记录复合语句的类型、入口标签和出口标签。只要翻译`break`语句时栈中存在循环或`switch`复合语句，`break`语句便是合法的，否则报告语义错误。

`continue`语句与`break`语句类似，不过翻译`continue`语句时，只需要关心循环语句的嵌套层次。

`return`语句有两种形式：有返回值的`return`和无返回值的`return`。由于我们设计的函数限定返回值只能是基本类型，而基本类型只有`int`和`char`类型，它们是相互兼容的，因此`return`语句的语义分析是检查`return`语句和函数返回值类型分别为`void`及基本类型时的情况。

```

1 void GenIR::genReturn
  (Var*ret){
2     if(!ret)return;
3     Fun*fun=symtab.getCurFun();
4     if(ret->isVoid()&&fun->getType()!=KW_VOID
5         ||ret->isBase()&&fun->getType()==KW_VOID){           //类型不兼容
6
7         SEMERROR(RETURN_ERR
8             );
9             //return语句和函数返回值类型不匹配
10
11         return;
12     }
13     InterInst* returnPoint=fun->getReturnPoint();           //获取返回点
14
15     if(ret->isVoid())
16         symtab.addInst(new InterInst(OP_RET,returnPoint));
17     else{
18         if(ret->isRef())ret=genAssign(ret);                   //处理
19         ret是
20         *p情况
21
22         symtab.addInst(new InterInst(OP_RETV,returnPoint,ret));
23     }
24 }

```

代码的第4~8行对return语句的返回类型进行检查。如果函数返回值不是void，那么return语句返回值类型也不能是void。如果函数返回值是void，return语句返回值类型也必须是void。否则，报告语义错误。

第9~15行为return语句的翻译代码，详见3.5.3节描述的内容。

3.4.4 错误处理

在前面讨论语义分析时，语义错误都是使用**SEMERROR**宏进行处理，这与词法错误、语法错误的处理类似。不过语义分析除了报告语义错误，还包含语义警告**SEMWARN**，比如验证函数声明形参匹配时触发的语义警告。语义分析中处理的语义错误和语义警告如表3-7所示。

表3-7 语义错误与语义警告表

语义错误/语义警告类型	语义错误/语义警告信息
VAR_RE_DEF	变量重定义
FUN_RE_DEF	函数重定义
VAR_UN_DEC	变量未声明
FUN_UN_DEC	函数未声明
FUN_DEC_ERR	函数声明与定义不匹配
FUN_CALL_ERR	函数形参与实参不匹配
DEC_INIT_DENY	变量声明时不允许初始化
EXTERN_FUN_DEF	函数定义不能声明 extern
ARRAY_LEN_INVALID	数组长度应该是正整数
VAR_INIT_ERR	变量初始化类型错误
GLB_INIT_ERR	全局变量初始化值不是常量
VOID_VAR	变量不能声明为 void 类型
EXPR_NOT_LEFT_VAL	无效的左值表达式
ASSIGN_TYPE_ERR	赋值表达式类型不兼容
EXPR_IS_BASE	表达式操作数不能是基本类型
EXPR_NOT_BASE	表达式操作数不是基本类型
ARR_TYPE_ERR	数组索引运算类型错误
EXPR_IS_VOID	void 的函数返回值不能参与表达式运算
BREAK_ERR	break 语句不能出现在循环或 switch 语句之外
CONTINUE_ERR	continue 不能出现在循环之外
RETURN_ERR	return 语句与函数返回值类型不匹配
FUN_DEC_CONFLICT	函数参数列表类型冲突
FUN_RET_CONFLICT	函数返回值类型不精确匹配

在扫描器内，我们计算了字符的行的位置，还保存了处理的源文件名称。根据表3-7提供的语义错误/警告信息，实现语义错误/警告信息输出的相关代码如下：

```

1  /*
2   语义错误类型

3  */
4  enum SemError
5  {
6      VAR_RE_DEF,                //变量重定义
7
8      FUN_RE_DEF,                //函数重定义
9  }
```

```

7          VAR_UN_DEC,                                // 变量未声明

8          FUN_UN_DEC,                                // 函数未声明

9          FUN_DEC_ERR,                                // 函数声明与定义不匹配

10         FUN_CALL_ERR,                                // 形参与实参不匹配

11         DEC_INIT_DENY,                                // 声明不允许初始化

12         EXTERN_FUN_DEF,                                // 函数声明不能使用
extern
13         ARRAY_LEN_INVALID,                            // 数组长度无效

14         VAR_INIT_ERR,                                // 变量初始化类型错误

15         GLB_INIT_ERR,                                // 全局变量初始化值不是常量

16         VOID_VAR,                                    // void 变量

17         EXPR_NOT_LEFT_VAL,                            // 无效的左值表达式

18         ASSIGN_TYPE_ERR,                                // 赋值类型不匹配

19         EXPR_IS_BASE,                                // 表达式操作数不能是基本类型

20         EXPR_NOT_BASE,                                // 表达式操作数不是基本类型

21         ARR_TYPE_ERR,                                // 数组运算类型错误

22         EXPR_IS_VOID,                                // 表达式不能是
VOID类型

23         BREAK_ERR,                                    // break不在循环或
switch-case中

24         CONTINUE_ERR,                                // continue不在循环中

25         RETURN_ERR                                    // return语句与函数返回值
类型不匹配

26 };
27 /*
28  语义警告类型

```

```

29 */
30 enum SemWarn

{
31     FUN_DEC_CONFLICT, //函数参数列表类型冲突

32     FUN_RET_CONFLICT //函数返回值类型冲突

33 };
34 /*
35 打印语义错误

36 */
37 void Error::semError

(int code,string name){
38     static const char *semErrorTable[]={
39         "变量重定义

",
40         "函数重定义

",
41         "变量未声明

",
42         "函数未声明

",
43         "函数声明与定义不匹配

",
44         "函数形参与实参不匹配

",
45         "变量声明时不允许初始化

",
46         "函数定义不能使用声明保留字

extern",
47         "数组长度应该是正整数

",
48         "变量初始化类型错误

",
49         "全局变量初始化值不是常量

",
50         "变量不能声明为

void类型

",
51         "无效的左值表达式

",
52         "赋值表达式类型不兼容

",
53         "表达式操作数不能是基本类型

```

```

"/
54          "表达式操作数不是基本类型

"/
55          "数组索引运算类型错误

"/
56          "void的函数返回值不能参与表达式运算

"/
57          "break语句不能出现在循环或
switch语句之外

"/
58          "continue不能出现在循环之外

"/
59          "return语句和函数返回值类型不匹配

"
60      };
61      errorNum++;
62      printf("%s<第

%d行

> 语义错误

: %s %s.\n",
63         scanner->getFile(),
64         scanner->getLine(),
65         name.c_str(),
66         semErrorTable[code]);
67     }
68     /*
69         打印语义警告

70 */
71 void Error::semWarn

(int code,string name)
72 {
73     //语义警告信息串

74     static const char *semWarnTable[]={
75         "函数参数列表类型冲突

"/
76         "函数返回值类型不精确匹配

"
77     };
78     warnNum++;
79     printf("%s<第

%d行

> 语义警告

: %s %s.\n",
80         scanner->getFile(),
81         scanner->getLine(),
82         name.c_str(),

```

```
83             semWarnTable[code]);  
84 }  
85 #define SEMERROR  
  
(code,name) Error::semError(code,name)  
86 #define SEMWARN  
  
(code,name) Error::semWarn(code,name)
```

第1~33行使用枚举类型**SemError**和**SemWarn**记录了所有语义错误/警告的类型。

第34~67行定义输出语义错误信息的函数**semError**，其中数组**semErrorTable**保存了与语义错误类型对应的信息。第61行使用变量**errorNum**记录编译器产生的错误数。第62~67行调用了扫描器的方法获取语义错误产生位置所在的文件名和行号。

第68~84行定义输出语义警告信息的函数**semWarn**，其中数组**semWarnTable**保存了与语义警告类型对应的信息。第78行使用变量**warnNum**记录编译器产生的警告数。第79~84行调用了扫描器的方法获取语义错误产生位置所在的文件名和行号。

第85行使用宏**SEMERROR**封装**semError**函数的调用。

第86行使用宏**SEMWARN**封装**semWarn**函数的调用。

至此，我们描述了语义分析实现的细节。由于语义分析穿插在符号表管理和代码生成器的实现中，虽然在符号表管理中描述了所有相

关的语义分析细节，但是代码生成涉及的语义分析并未详尽描述。在接下来的代码生成章节中，会对相关的语义分析做进一步阐述。

3.5 代码生成

源代码经过词法分析、语法分析、语义分析后，如果未产生错误，则通过语法模块相应的语义动作进行代码生成。代码生成本质上是将语法模块翻译为汇编代码，我们可以选择在识别每个语法模块时，直接输出对应的汇编代码。

例如对于全局变量a和b，赋值语句“a=b;”便可以翻译为：

```
mov eax,[b]
mov [a],eax
```

根据赋值表达式的文法定义，我们在赋值表达式的递归下降子程序内插入代码生成对应的语义动作代码。

```
1  //<asexpr>-><orexpr><asstail>
2  Var* Parser::asexpr
   (){
3      Var*lval=orexpr
   ();
4      return asstail
   (lval);
5  }
6  //<asstail>->ASSIGN <orexpr><asstail> | ε

7  Var* Parser::asstail
   (Var*lval){
8      if(match(ASSIGN
   )){
9          Var*val=orexpr
   ();
```

```

10          Var*rval=asstail
(val);
11          Var*result=ir.genTwoOp
(lval,ASSIGN,rval);
12          return result;
13      }
14      return lval;
15 }
16 Var* GenIR::genTwoOp
(Var*lval,Tag opt,Var*rval){
17     if(opt==ASSIGN){
18         fprintf(file,"mov eax,[%s]\n",rval->getName().c_str());
19         fprintf(file,"mov [%s],eax\n",lval->getName().c_str());
20         return lval;
21     }
22 }

```

在函数`asstail`中，我们考虑到了赋值运算符的右结合性质，因此在第11行，将赋值语句代码生成的函数调用`genTwoOp`放在了第10行`asstail`调用之后，这样就可以保证赋值表达式从右向左计算。如果表达式的运算符是左结合的，那么只需要调换第10、11行代码的位置，并修改相应的参数即可，例如：

```

10 Var*result=ir.genTwoOp(lval,ASSIGN,val);
11 Var*rval=asstail(result);
12 return rval;

```

从这里也可以看出运算符的结合性只是影响了代码生成动作的位置，而不会影响文法产生式的结构。

第16~21行描述了赋值表达式的代码生成片段，即生成上述两条汇编语句。当然，赋值表达式的返回值应该被赋值的变量`lval`。然而，这里只给出了全局变量到全局变量赋值表达式的翻译。由于变量存储

的多样性，比如局部变量使用[ebp+栈帧偏移]的方式访问，而常量则直接使用立即数访问，这样的翻译方式考虑的组合情况就太多了。

另外，直接将代码的翻译工作硬编码虽然可以完成代码生成，但是当需要生成其他目标指令集的汇编代码或者对代码进行优化时就非常困难。因此，需要设计一个中间层来屏蔽具体机器的指令集细节，同时避免考虑代码生成时变量存储访问的复杂性。

如图3-22所示，代码生成器根据语义动作，并非直接将语法模块翻译为x86汇编代码，而是生成中间代码。中间代码生成后，再将中间代码转换为具体机器指令集的汇编代码。如果编译器支持代码优化，优化器会对中间代码和目标代码进行处理，生成更高效的目标代码。

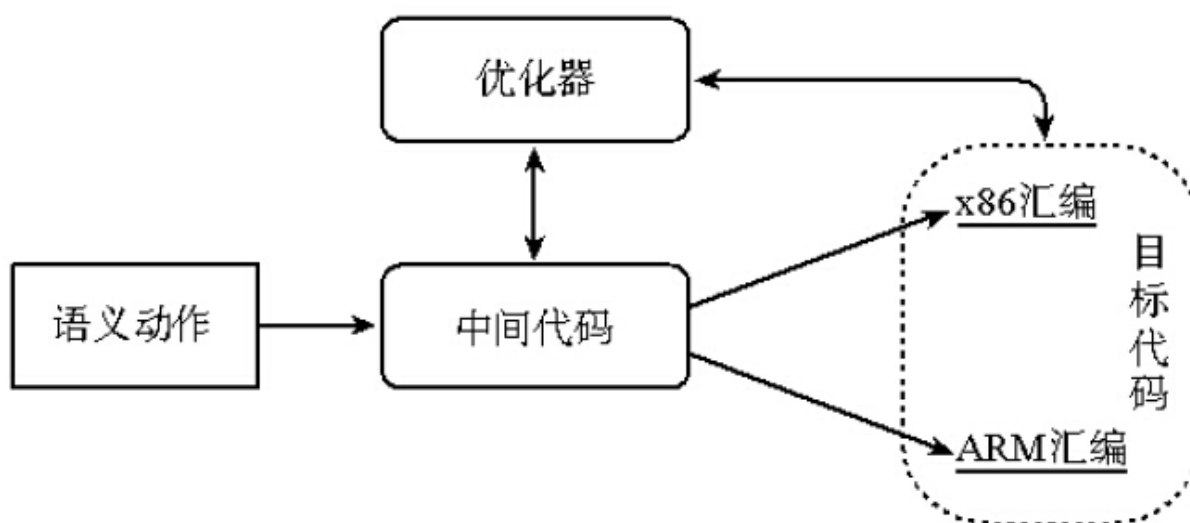


图3-22 代码生成器结构

3.5.1 中间代码设计

编译器使用的中间代码形式有多种，比如抽象语法树形式、三元式、四元式、静态单赋值形式等，本书采用四元式作为中间代码形式。

四元式包含四个基本字段：操作符`op`、运算结果`result`、第一个操作数`arg1`、第二个操作数`arg2`，其一般形式为：

<code>result = arg1 op arg2</code>

四元式的含义为使用操作符`op`得到操作数`arg1`和`arg2`的计算结果，并将结果保存到`result`中。

如表3-8所示，并非每一条中间代码指令都使用了四元式的所有字段，比如操作符`op_neg`表示取负运算，未使用`arg2`字段。甚至某些字段具有多重含义，比如操作符`op_proc`表示无返回值函数调用，`arg1`字段在表达式中一般表示一个变量对象，而在这里表示被调用的函数对象。

表3-8 中间代码指令

op	result	arg1	arg2	含义
op_label	L	-	-	定义标签 L
op_entry	-	f	-	函数 f 入口
op_exit	-	f	-	函数 f 出口
op_dec	-	s	-	声明符号 s
op_as	x	y	-	x=y
op_add	x	y	z	x=y+z
op_sub	x	y	z	x=y-z
op_mul	x	y	z	x=y*z
op_div	x	y	z	x=y/z
op_mod	x	y	z	x=y%z
op_neg	x	y	-	x=-y
op_gt	x	y	z	x=(y > z)
op_ge	x	y	z	x=(y >= z)
op_lt	x	y	z	x=(y < z)
op_le	x	y	z	x=(y <= z)
op_equ	x	y	z	x=(y == z)
op_ne	x	y	z	x=(y != z)
op_not	x	y	-	x=!y
op_and	x	y	z	x=(y && z)
op_or	x	y	z	x=(y z)
op_lea	x	y	-	x=&y
op_set	x	y	-	*y=x
op_get	x	y	-	x=*y
op_jump	L	-	-	goto L
op_jt	L	s	-	if(s)goto L
op_jf	L	s	-	if(!s)goto L
op_arg	-	s	-	传入参数 s
op_proc	-	f	-	调用 fun()
op_call	x	f	-	x=fun()
op_ret	-	-	-	return
op_retv	-	s	-	return s

中间代码指令的操作符的定义为:

```
1 enum Operator
```

```
{
```

```
2     OP_NOP,
```

```
// 空指令
```

```

3      OP_DEC,                                     //声明
4      OP_ENTRY, OP_EXIT,                         //函数出入口
5      OP_AS,                                     //赋值
6      OP_ADD, OP_SUB, OP_MUL, OP_DIV, OP_MOD,    //算术运算
7      OP_NEG,                                    //取负
8      OP_GT, OP_GE, OP_LT, OP_LE, OP_EQU, OP_NE, //关系运算
9      OP_NOT,                                    //非
10     OP_AND, OP_OR,                             //与、或
11     OP_LEA,                                    //取址
12     OP_SET, OP_GET,                            //指针运算
13     OP_JMP,                                    //无条件跳转
14     OP_JT, OP_JF, OP_JNE,                     //条件跳转
15     OP_ARG,                                    //参数传递
16     OP_PROC,                                   //调用过程
17     OP_CALL,                                   //调用函数
18     OP_RET,                                    //直接返回
19     OP_RETV                                    //带数据返回
20 };

```

其中操作符标签都使用大写形式，**OP_NOP**指令作为操作符的默认值。

我们使用**InterInst**数据结构描述一条中间代码指令。

```
1 class InterInst
{
2     string label;           // 标签

3     Operator op;           // 操作符

4     Var *result;           // 运算结果

5     Var*arg1;               // 参数
1
6     Var *arg2;              // 参数
2
7     Fun*fun;                // 函数

8     InterInst*target;       // 跳转标号

9 };
```

字段label表示当前指令是标签还是真正的指令。如果label为空串，则表示正常的指令，否则记录标签的名称。

字段op、result、arg1和arg2分别记录了四元式的基本要素。

字段fun记录某些四元式使用的函数对象，比如op_proc指令调用的函数。

字段target记录某些四元式使用的标签，比如op_jump指令的目标标签。

在代码生成器中，语法模块的翻译结果不再是目标指令集的汇编代码，而是中间代码指令序列。在Fun对象中的interCode字段是vector<InterInst*>类型，它记录了函数的中间代码指令序列。当所有的

代码被编译器处理完毕后，每个interCode都保存了对应函数的中间代码翻译结果，此时只需要将这些中间代码再翻译为目标指令集的汇编代码即可。

3.5.2 程序运行时存储

在代码翻译的过程中，涉及很多程序运行时的细节。不同的操作系统和指令集体系结构，其代码翻译方式也不尽相同。因此在讨论具体语法模块的代码生成之前，我们需要了解程序运行时存储相关的知识。

1. 进程内存组织

我们知道在操作系统中，程序运行时是以进程形式存在的。在高级语言层面，我们面对的是变量、函数、表达式和语句等语言元素，而在进程的内存空间中，所有的高级语言信息都转化为二进制形式。弄清高级语言元素与进程、内存占用情况的对应关系，对代码生成至关重要。

如图3-23所示，在32位的Linux系统中，进程拥有4GB的线性地址空间。其中高1GB的地址空间分配给操作系统内核，剩余的3GB地址空间称为用户空间，用于存放进程的用户级代码和数据。在用户空间中，最为关键的部分为代码段、数据段、堆和栈。

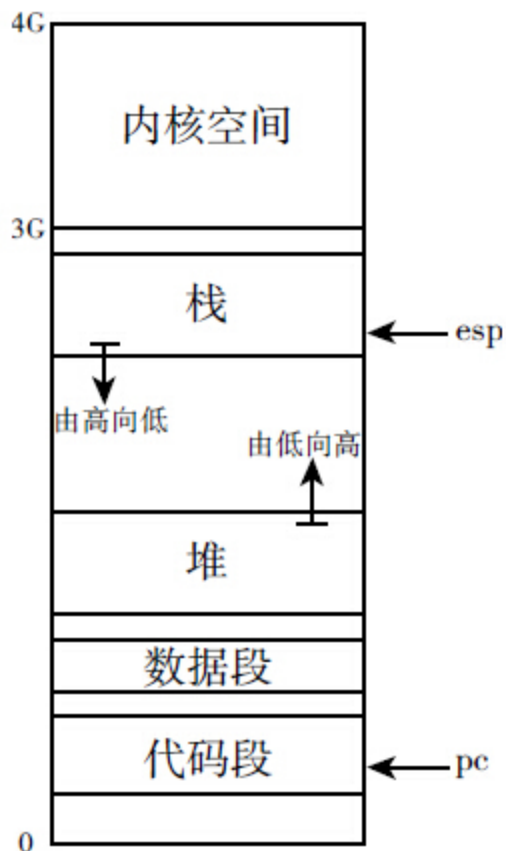


图3-23 进程内存组织

其中，代码段“`.text`”保存进程的二进制代码，其中CPU的程序计数器`pc`指针就是指向代码段区的某条指令的起始地址。程序执行时，`pc`指针在代码段内移动，实现程序的顺序、跳转执行。

数据段“`.data`”保存进程的静态数据，比如全局变量、常量字符串等。在Linux系统中，一般将初始化的全局变量放在数据段，将未初始化的全局变量放在“`.bss`”段，而将常量字符串放在具有只读属性的“`.rodata`”段内（也称文字池）。为了降低编译系统实现的复杂度，我们将所有的数据放在数据段内。

进程空间内的堆并非数据结构学科中所描述的最大（小）堆，它是保存程序运行时动态分配的数据所在的地址空间。C语言的**malloc**和**free**操作的内存便是堆的内存。当堆空间不足时，操作系统会自动增加堆的大小，堆的地址空间是向高地址方向增长的。我们设计的自定义语言不涉及动态内存分配，因此不会使用堆空间。

进程空间内的栈与数据结构理论中所描述的栈比较相似，它满足先进后出的原则，而且是一块连续的内存空间。CPU的栈指针寄存器**esp**总是指向栈顶位置，当数据入栈时，**esp**减小，当数据出栈时，**esp**增加。当栈空间不足时，操作系统会自动增加栈的大小，栈的地址空间是向低地址方向增长的。栈在程序语言的功能实现中十分重要，函数调用、实际参数传递以及局部变量的存储都是由栈来完成。

在代码生成中，需要明确每次语法模块的翻译所涉及的进程内存。源代码定义的全局变量和字符串常量需要保存到数据段，函数体内的表达式计算和复合语句以二进制代码的形式保存到代码段，函数调用的实际参数和函数内的局部变量保存在栈内。

2.函数栈帧管理

从进程内存空间的角度来看，函数调用是一个复杂的过程。它牵涉到代码段中函数定义、调用和返回的二进制代码。同时，参数的传递、局部变量的管理都与栈息息相关。一般使用栈帧描述一次函数的

调用过程，当发生函数调用时，需要开辟栈帧保存实际参数、局部变量等信息，当函数调用结束时，需要将函数的栈帧释放。

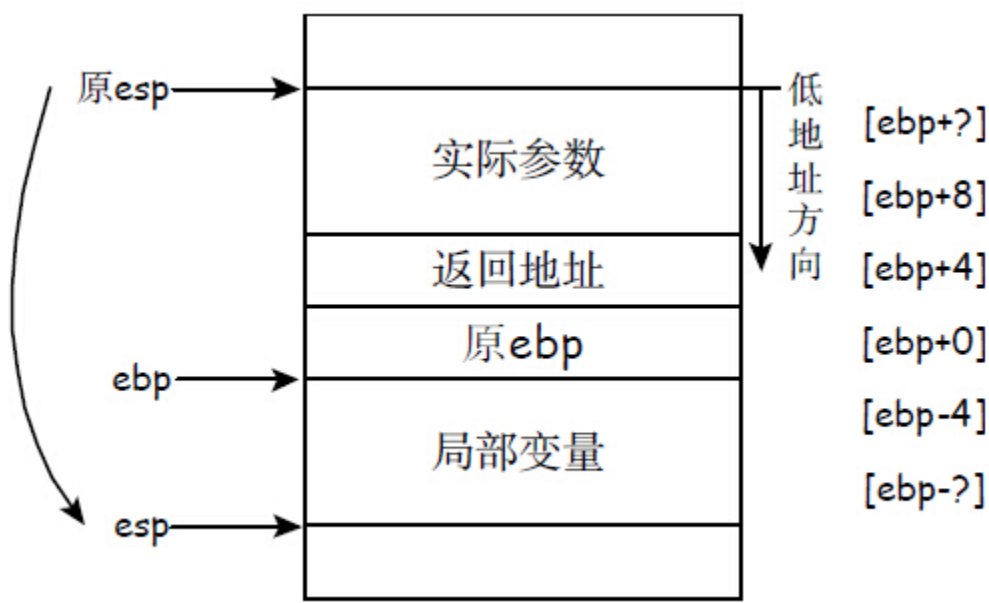


图3-24 函数栈帧

如图3-24所示，函数调用前，栈顶在“原esp”指向的位置。函数调用发生时，依次入栈的数据为实际参数、函数返回地址、ebp寄存器值和局部变量。

根据C语言函数调用规则，实际参数从右向左依次入栈。假设函数调用形式为：

```
fun(0,1);
```

使用push指令将实际参数入栈。

```
push 1
push 0
```

实际参数入栈完毕后，使用call指令进行函数调用。

```
call fun
```

指令call fun的含义为，将call指令的下一条指令的起始地址（函数调用后的返回地址）入栈，然后跳转到fun标签的位置继续执行。

跳转到fun标签后，便开始执行fun函数的指令。最先执行的两条指令是：

```
push ebp
mov ebp, esp
```

其中，第一条指令是将ebp寄存器值入栈。第二条指令是将栈指针esp保存到ebp。这样ebp指向的内存保存的内容恰好是原ebp寄存器的值。而实际参数相对于ebp寄存器的偏移依次为8、12、16……（假设实际参数都是4字节大小。）

实际参数

0:

[ebp+8] 实际参数

1:

[ebp+12]

设定好ebp寄存器后，接下来为局部变量开辟栈空间，假如fun函数有两个int类型的局部变量。

```
sub esp,8
```

这样栈指针esp便指向了最后一个局部变量，而局部变量相对于ebp寄存器的偏移依次为-4、-8、-12……（假设局部变量都是4字节大小。）

局部变量

1:

[ebp-4]局部变量

2:

[ebp-8]

一般称ebp为函数栈帧的基址，对实际参数和局部变量的访问都是对基址ebp加偏移的内存访问。由于每次函数调用时产生的函数栈帧都有自己的ebp值，因此每次函数调用都需要首先将ebp寄存器入栈保存。

函数调用结束后，需要将函数栈帧释放。首先处理函数的返回值，假设fun函数返回值为0。

```
mov eax,0
```

根据C语言函数调用规则，函数调用后，返回值保存在寄存器**eax**中。

然后释放局部变量的空间，恢复**ebp**寄存器的值。

```
mov esp,ebp  
pop ebp
```

第一条指令将栈指针**esp**恢复为先前保存在**ebp**寄存器中的值，然后使用**pop**指令恢复**ebp**的值。

最后使用**ret**指令实现函数返回。

```
ret
```

函数返回指令**ret**的含义为，取出栈顶保存的函数返回地址，然后跳转到该地址继续执行。

函数返回后，栈指针**esp**并未恢复到“原**esp**”的位置，因此函数调用者需要显式地释放实际参数的空间。

```
add esp,8
```

由于调用函数**fun**时，压入了两个**4**字节参数到栈中，因此需要将**esp**指针加上**8**字节，恢复到调用函数之前的状态。

在整个函数调用的过程中，我们称从实际参数到局部变量所开辟的内存空间为函数栈帧。而函数栈帧基址由ebp寄存器保存，对实际参数和局部变量的访问都是通过ebp的基址寻址来实现。

根据函数的栈帧管理方式，来决定函数定义代码和调用代码的翻译。假设有如下函数定义：

```
1 int fun(int x){
2     int y=x;
3     return y;
4 }
```

将该函数定义翻译为x86汇编代码形式为：

```
1 fun:
2     push
3         ebp
4         mov
5             esp, ebp
6         sub
7             esp, 4
8         mov
9             eax, [ebp+8]
10        mov
11            [ebp-4], eax
12        mov
13            eax, [ebp-4]
14        mov
15            esp, ebp
16        pop
17            ebp
18        ret
```

第1行为函数名标签**fun**，它记录函数开始执行的位置。

第2~3行为进入函数代码，用于保存**ebp**，设定新的栈帧基址。

第4行为局部变量开辟栈帧空间。

第5~6行通过**ebp**的基址寻址访问参数变量和局部变量，完成函数内代码的翻译。

第7行将函数返回值保存到**eax**寄存器中。

第8~10行为退出函数代码，恢复栈指针**esp**、栈基址**ebp**，并执行函数返回。

对于函数调用者，假设函数调用形式为：

```
glb=fun(0);
```

假定**glb**变量为全局变量，那么函数调用翻译为x86汇编的形式为：

```
1 push 0
2 call fun
3 add esp,4
4 mov [glb],eax
```

其中**push**指令将参数0入栈，然后使用**call**指令调用函数**fun**。 **fun**调用结束后，需要释放实际参数的栈空间，因此需要将**esp**加4。最后将

保存在**eax**的函数返回值传送给全局变量**glb**。

3.5.3 函数定义与return语句翻译

明确了函数栈帧管理的方式后，函数定义的代码翻译就比较简单了。在前面描述的符号表的defFun函数实现中，调用了genFunHead产生函数的入口代码。而在符号表的endDefFun函数实现中，调用了genFunTail产生函数的出口代码。

```
1 void GenIR::genFunHead
  (Fun*function){
2     function->enterScope();
  //进入函数

3     symtab.addInst(new InterInst(OP_ENTRY
  ,function));          //函数入口

4     function->setReturnPoint(new InterInst);          //创
  建返回点

5 }
6 void GenIR::genFunTail
  (Fun*function){
7     symtab.addInst(function->getReturnPoint());          //函数返回
  点

8     symtab.addInst(new InterInst(OP_EXIT
  ,function));          //函数出口

9     function->leaveScope();
  //退出函数

10 }
```

第1~5行为genFunHead实现代码，首先调用Fun的enterScope进入函数作用域，然后创建中间代码指令OP_ENTRY提供函数入口，并使用符号表提供的addInst函数将该指令添加到函数对象的interCode内。最后使用无参InterInst构造函数创建一个唯一的标签，保存到函数对象的returnPoint字段。

第6~10行为genFunTail实现代码，首先将函数对象的returnPoint指向的标签添加到interCode，然后创建中间代码指令OP_EXIT以提供函数出口，并添加到interCode。最后调用leaveScope退出函数作用域。

在上述代码中，总是涉及函数对象的字段returnPoint。该字段指向一个InterInst对象，表示函数退出代码的位置。之所以保存这个字段，是为了翻译return语句的需要。

假设函数体内出现了多个return指令。

```
1  int fun(){
2      return 1;
3      ...
4      return 0;
5  }
```

由于每次return语句执行后都需要执行函数退出代码，为了避免每次翻译return语句都重复产生函数退出代码，因此使用returnPoint记录退出代码的位置，每次return语句执行后只需要跳转到该位置即可。我们希望fun函数翻译为中间代码形式如下：

```
1 fun:
2     OP_ENTRY fun
3     OP_RETV 1 goto returnPoint
4     ...
5     OP_RETV 0 goto returnPoint
6 returnPoint:
7     OP_EXIT fun
```

根据前面的要求，return语句的翻译如下：

```
1 void GenIR::genReturn
  (Var*ret){
2     if(!ret)return;
3     Fun*fun=symtab.getCurFun();
4     if(ret->isVoid()&&fun->getType()!=KW_VOID
5         ||ret->isBase()&&fun->getType()==KW_VOID){
//类型不兼容

6         SEMERROR(RETURN_ERR);                                //return语句返回值与函数返回类
//型不匹配

7         return;
8     }
9     InterInst* returnPoint=fun->getReturnPoint
  ();                                //获取返回点

10    if(ret->isVoid())
11        symtab.addInst(new InterInst(OP_RET
,returnPoint));
12    else{
13        if(ret->isRef())ret=genAssign(ret);
//处理
ret是
*p情况

14        symtab.addInst(new InterInst(OP_RETV
,returnPoint,ret));
15    }
16 }
```

第2~8行为语义分析的代码，前面已经描述过。

第9行取出函数对象的returnPoint字段。

第10~11行判断函数返回值变量为**void**类型，因此产生中间代码指令OP_RET。

第12~15行判断函数具有返回值，因此产生中间代码指令OP_RETV。

其中第13行处理形如“**return***p;”的返回语句，在指针运算的翻译中会对此做详细描述。

我们根据返回值变量的类型来决定**return**语句的翻译结果，这里涉及**void**类型的返回值变量。

```
1 void Parser::statement
  (){
2     switch(look->tag)
3     {
4         ...
5         case KW_RETURN:
6             move();
7             ir.genReturn
            (altexpr());           //产生
            return语句

8
9             ...
10            break;
11        ...
12    }
13 Var* Parser::altexpr

14    (){
15        if(EXPR_FIRST)
16            return expr();
17        return Var::getVoid

18    };           //返回特殊
19    void变量

20 }

21 }
```

根据statement子程序对return语句的处理，genReturn函数的参数为altexpr子程序的返回值。在altexpr子程序中，对于空表达式返回void类型变量。

3.5.4 表达式翻译

在高级语言程序中，程序真正的计算操作是由各种表达式计算完成的，因此表达式可以理解为程序计算的抽象。由于表达式的结构具有相似性，因此表达式的翻译可以分类处理。我们将表达式分为5类：双目运算表达式、前缀单目运算表达式、后缀单目运算表达式、数组索引表达式和函数调用表达式。

1. 指针运算的处理

指针运算使表达式的翻译复杂化，因此在讨论其他表达式的翻译前，我们首先处理指针运算表达式的代码生成。

```
1  Var* GenIR::genPtr
   (Var*val){
2      if(val->isBase()){
3          SEMERROR(EXPR_IS_BASE);                //基本类
   型不能取值

4          return val;
5      }
6      Var*tmp=new Var(symtab.getScopePath()
7          ,val->getType(),false);
8      tmp->setLeft(true);                          //指针运算结
   果为左值

9      tmp->setPointer
   (val);                //设置指针变量

10     symtab.addVar(tmp);
11     return tmp;
12 }
```

第2~5行实现指针运算的语义分析，对于基本类型的变量，是不能进行指针运算的。

第6行创建与指针变量`val`的类型相同的基本类型变量`tmp`。

指针运算的结果可以作为左值，即可以取址或者被赋值。因此第8行将`tmp`设置为左值。

第9行将指针变量`val`记录到`tmp`的`ptr`字段，表示`tmp`变量来源于对`val`变量的指针运算结果。

第10~11行将`tmp`添加到符号表，并返回。

我们发现，处理指针运算表达式时，并未产生任何中间代码。这是因为指针运算的特殊性。假设变量`x`为`int`类型的变量，`p`为`int`类型的指针变量。

```
1 x=*p;  
2 *p=x;
```

对于语句1，我们可以翻译为如下中间代码：

```
OP_GET tmp p  
OP_AS x tmp
```

中间代码指令`OP_GET`指令将`p`指针指向的内存中的内容复制到`tmp`，然后`OP_AS`将`tmp`的内容复制到`x`。

而对于语句2，应该翻译为如下中间代码：

```
OP_SET x p
```

中间代码指令OP_SET将x的值复制到p指针指向的内存，而与genPtr中产生的临时变量tmp无任何关系！

由此，我们可以得出一个结论：如果将指针运算的结果tmp作为右值，那么可以使用tmp参与表达式的翻译，如果tmp作为左值使用，那么仍使用原来的指针变量参与表达式的翻译。

通过取址运算表达式的翻译，可以清晰地看出这一点。

```
1  Var* GenIR::genLea
   (Var*val){
2      if(!val->getLeft()){
3          SEMERROR(EXPR_NOT_LEFT_VAL);
   //不能取地址

4          return val;
5      }
6      if(val->
isRef
   ()) //类似
   &*p运算

7          return val->getPointer(); //取出变量的指针

8      else{
   //一般取址运算

9          Var* tmp=new Var(symtab.getScopePath()
10             ,val->getType(),true); //产生局部变量
   tmp
```



```

11          symtab.addVar(tmp);
12          symtab.addInst(new InterInst(OP_LEA
, tmp, val));
13          return tmp;
14      }
15 }

```

第2~5行实现取址运算的语义分析，不能对非左值变量做取址操作。

第6行判断变量`val`是否是指针运算的结果，`isRef`函数表示变量对象的`ptr`是否有效。如果`val`是指针运算的结果，则直接返回变量的`ptr`字段。

第8~14行处理`val`是普通变量对象的情况。使用中间代码操作符`OP_LEA`将`val`变量的地址取出复制到变量`tmp`，并返回。

除了对指针运算的结果进行取址运算外，其他表达式运算也需要考虑操作数是否是指针运算结果的情况，因为指针运算结果变量与普通的基本类型变量在运算特性上完全等价。

```

1  Var* GenIR::genAssign
(Var*val){
2      Var*tmp=new Var(symtab.getScopePath(),val);           //复制变量信息

3      symtab.addVar(tmp);
4      if(val->isRef())
5          symtab.addInst(new InterInst(OP_GET
6              , tmp, val->getPointer()));
7      else
8          symtab.addInst(new InterInst(OP_AS
9              , tmp, val));
10     return tmp;
11 }

```

我们使用单参数的genAssign函数处理指针运算的结果变量作为右值的情况。第3行判断val如果是指针运算结果，那么就使用OP_GET指令产生val变量的ptr字段到tmp的指针运算。否则，产生变量val到tmp的赋值运算。

```
1  Var* GenIR::genAssign
   (Var*lval,Var*rval){
2      if(!lval->getLeft()){
3          SEMERROR(EXPR_NOT_LEFT_VAL);
4          return rval;
5      }
6      if(!typeCheck
   (lval,rval)){
7          SEMERROR(ASSIGN_TYPE_ERR);
8          return rval;
9      }
10     if(rval->isRef()) //处理
   右值
   (*p)
11         rval=genAssign
   (rval);
12     if(lval->isRef()) //处理
   左值
   (*p)
13         symtab.addInst(new InterInst(OP_SET
   ,rval,lval->getPointer()));
14     else
15         symtab.addInst(new InterInst(OP_AS
   ,lval,rval));
17     return lval;
18 }
```

指针运算结果作为左值被赋值的情况由双参数genAssign函数处理。

第2~5行检查被赋值的变量是否是左值，第6~9行使用typeCheck函数检查赋值表达式的两个操作数类型是否兼容。

第10~11行处理赋值表达式的右值rval，如果rval是指针运算结果，则使用单操作数genAssign取出rval的值。

第12~16行处理赋值表达式的左值lval，如果lval是指针运算结果，则产生rval到lval的ptr的指针赋值运算。否则，产生rval到lval的赋值运算。

函数typeCheck的实现如下：

```
1  bool GenIR::typeCheck
   (Var*lval,Var*rval){
2      bool flag=false;
3      if(!rval)return false;
4      if(lval->isBase()&&rval->isBase())                //都是基本类型

5          flag=true;
6      else if(!lval->isBase() && !rval->isBase())        //都不是基本类型

7          flag=rval->getType()==lval->getType();        //只要求类型相同

8      return flag;
9  }
```

第4行检查两个变量如果都是基本类型（int或char），则认为类型兼容，我们认为char和int类型的变量可以默认转换。

第6~7行检查两个变量如果都不是基本类型，则检查变量类型type字段是否相同，如果相同则兼容。除此之外的情况，变量的类型不兼

容。

2.双目运算表达式

表达式运算中，双目运算表达式占据大多数，比如前面讨论的赋值表达式就是双目运算表达式。其他双目运算表达式还有逻辑运算（与、或），关系运算（大于、大于等于、小于、小于等于、等于、不等于），算术运算（加、减、乘、除、取模）表达式。对双目运算表达式的翻译实现如下：

```
1  Var* GenIR::genTwoOp
   (Var*lval,Tag opt,Var*rval){
2      if(!lval || !rval)return NULL;
3      if(lval->isVoid()||rval->isVoid()){
4          SEMERROR(EXPR_IS_VOID);
5          return NULL;
6      }
7      if(opt==ASSIGN)return genAssign
   (lval,rval);           //赋值

8      if(lval->isRef())lval=genAssign(lval);           //处理左值

9      if(rval->isRef())rval=genAssign(rval);           //处理右值

10     if(opt==OR)return genOr
   (lval,rval);           //或

11     if(opt==AND)return genAnd
   (lval,rval);           //与

12     if(opt==EQU)return genEqu
   (lval,rval);           //等于

13     if(opt==NEQU)return genNequ
   (lval,rval);           //不等于
```

```

14      if(opt==ADD)return genAdd
(lval,rval);                                //加

15      if(opt==SUB)return genSub
(lval,rval);                                //减

16      if(!lval->isBase() || !rval->isBase())
17      {
18          SEMERROR(EXPR_NOT_BASE);
19          return lval;
20      }
21      if(opt==GT)return genGt
(lval,rval);                                //大于

22      if(opt==GE)return genGe
(lval,rval);                                //大于等于

23      if(opt==LT)return genLt
(lval,rval);                                //小于

24      if(opt==LE)return genLe
(lval,rval);                                //小于等于

25      if(opt==MUL)return genMul
(lval,rval);                                //乘

26      if(opt==DIV)return genDiv
(lval,rval);                                //除

27      if(opt==MOD)return genMod
(lval,rval);                                //取模

28      return lval;
29 }

```

第3~6行处理表达式操作数为void类型变量的情况。

第7行单独处理赋值运算表达式，从前面讨论的genAssign的实现中可以看出，当被赋值的变量是指针运算结果时，需要使用OP_SET指令实现代码翻译。

第8~9行处理操作数为指针运算结果的情况，使用单参数的genAssign函数将指针运算结果取出。

第10~15行处理非基本类型可以参与的运算，包括或、与、等于、不等于、加和减运算。第21~27行处理只有基本类型可以参与的运算，包括大于、大于等于、小于、小于等于、乘和除运算。基本类型只能是int或char类型，除此之外的变量都是非基本类型，比如int*、char[]等。

在除了赋值运算表达式的其他双目运算表达式中，加法运算和减法运算表达式的翻译较为特殊。

```
1  Var* GenIR::genAdd
   (Var*lval,Var*rval)
2  {
3      Var*tmp=NULL;
4      if(!lval->isBase()&&rval->isBase()){
5          tmp=new Var(symtab.getScopePath(),lval);
6          rval=genMul
           (rval,Var::getStep
              (lval));
7      }
8      else if(lval->isBase()&&!rval->isBase()){
9          tmp=new Var(symtab.getScopePath(),rval);
10         lval=genMul
            (lval,Var::getStep
               (rval));
11     }
```

```

12     else if(lval->isBase()&&rval->isBase())
13         tmp=new Var(symtab.getScopePath(),KW_INT,false);
14     else{
15         SEMERROR(EXPR_NOT_BASE);
16         return lval;
17     }
18     symtab.addVar(tmp);
19     symtab.addInst(new InterInst(OP_ADD
, tmp, lval, rval));
20     return tmp;
21 }

```

第4~7行处理非基本类型操作数加上基本类型操作数的情况。例如指针变量`int*p`，在计算“`p+1`”时，实际的计算为“`p+1*sizeof(int)`”，因此通过调用乘法运算的翻译函数`genMul`将`rval`修改为实际累加的值。

第8~11行处理与“`p+1`”相似的情况“`1+p`”，同样的需要通过调用乘法运算翻译函数`genMul`将`lval`修改为实际累加的值。

第12~13行处理基本类型的操作数相加的情况。

第14~17行处理两个非基本类型的操作数相加，这种运算不合法，报告语义错误。

第19行，将加法运算翻译为`OP_ADD`指令表达式。

变量对象的`getStep`函数用于计算对变量加1操作时的实际累加值。

```

1  Var* Var::getStep
(Var*v){
2      if(v->isBase())return SymTab::one
;
3      else if(v->type==KW_CHAR)return SymTab::one

```

```
;
4      else if(v->type==KW_INT)return SymTab::four
;
5      else return NULL;
6  }
```

第2行处理基本类型操作数的加1运算，实际累加值仍为1。

第3行处理char*或char[]类型操作数的加1运算，实际累加值也是1。

第4行处理int*或int[]类型操作数的加1运算，实际累加值为4。

减法运算表达式的翻译与加法类似，不过限制更多。

```
1  Var* GenIR::genSub
   (Var*lval,Var*rval){
2      Var*tmp=NULL;
3      if(!rval->isBase())
4      {
5          SEMERROR(EXPR_NOT_BASE);
6          return lval;
7      }
8      if(!lval->isBase()){
9          tmp=new Var(symtab.getScopePath(),lval);
10         rval=genMul
   (rval,Var::getStep
   (lval));
11     }
12     else
13         tmp=new Var(symtab.getScopePath(),KW_INT,false);
14     symtab.addVar(tmp);
15     symtab.addInst(new InterInst(OP_SU
   B,tmp,lval,rval));
16     return tmp;
17 }
```

在加法运算中，对于指针变量`int*p`，可以执行“`p+1`”或“`1+p`”的操作。而减法运算中“`1-p`”的操作是非法的。

第3~7行处理`rval`不是基本类型的语义错误。

第8~11行处理`lval`不是基本类型的情况，将`rval`修正为实际的累加值。

第12~13行处理基本类型操作数的减法运算。

第15行将减法运算表达式翻译为`OP_SUB`指令。

除了已经讨论的赋值运算、加法运算和减法运算表达式，其他的双目运算表达式的翻译形式完全相同，比如乘法运算表达式。

```
1  Var* GenIR::genMul
   (Var*lval,Var*rval){
2      Var*tmp=new Var(symtab.getScopePath(),KW_INT,false);
3      symtab.addVar(tmp);
4      symtab.addInst(new InterInst(OP_MUL
   ,tmp,lval,rval));
5      return tmp;
6  }
```

对于其他双目运算表达式的实现，只是在创建`InterInst`对象时传递的操作符不同。由于自定义语言中的基本类型只有`int`和`char`，因此经过表达式运算后，如果运算结果仍为基本类型，一般我们都会使用`int`类型作为默认类型。

3.前缀单目运算表达式

前缀单目运算表达式包括取址，指针运算表达式，前缀自加、自减表达式，逻辑非运算和取负运算表达式。对单目运算表达式的翻译实现如下：

```

1 Var* GenIR::genOneOpLeft
(Tag opt,Var*val){
2     if(!val)return NULL;
3     if(val->isVoid()){
4         SEMERROR(EXPR_IS_VOID);
5         return NULL;
6     }
7     if(opt==LEA)return genLea
(val); //取址

8     if(opt==MUL)return genPtr
(val); //指针

9     if(opt==INC)return genIncl
(val); //自加

10    if(opt==DEC)return genDecL
(val); //自减

11    if(val->isRef())val=genAssign(val); //处理
(*p)
12    if(opt==NOT)return genNot
(val); //非

13    if(opt==SUB)return genMinus
(val); //取负

14    return val;
15 }

```

第3~6行处理表达式操作数为void类型变量的情况。

第7~8行处理取址和指针运算表达式，前面已经讨论过取址和指针运算表达式的翻译。

第9~10行处理前缀自加和自减运算表达式。

第12~13行处理逻辑非运算和取负运算表达式，这两种表达式运算的操作数是右值，因此第11行使用单参数的genAssign函数将可能的指针运算结果取出。

前缀自加和自减运算表达式的操作数是左值，因此可能是指针运算的结果。前缀自加运算表达式的翻译如下：

```
1  Var* GenIR::genIncl
   (Var*val){
2      if(!val->getLeft()){
3          SEMERROR(EXPR_NOT_LEFT_VAL);
4          return val;
5      }
6      if(val->isRef()){
//++*p
7          Var* t1=genAssign
   (val);                                //t1=*p
8          Var* t2=genAdd
   (t1,Var::getStep(val));                //t2=t1+1
9          genAssign
   (val,t2);                                //*p=t2
10         }
11         else
12             symtab.addInst(new InterInst(OP_ADD

/
13             val,val,Var::getStep(val)));                //++val
14         return val;
15     }
```

第2~5行产生操作数不是左值的语义错误。

第6~10行处理操作数是指针运算结果的情况。例如指针变量 `int*p`，对于表达式“`++*p`”的中间代码翻译结果应该如下：

<code>OP_GET t1 p</code>	<code>//t1=*p</code>
<code>OP_ADD t2 t1 1</code>	<code>//t2=t1+1</code>
<code>OP_SET t2 p</code>	<code>/*p=t2</code>

即首先使用处理指针运算结果`val`，使用单操作数`genAssign`将值取出到`t1`。然后使用`genAdd`对`t1`进行加1操作，结果保存到`t2`。最后使用双参数的`genAssign`函数将`t2`赋值到指针运算结果`val`。

第11~13行处理一般变量的加1操作，其中运算结果和第一个操作数都是`val`，第二个操作数是通过`getStep`计算的`val`实际累加值。

前缀自减运算表达式的翻译和前缀自加运算表达式相似，只需要将第8行的`genAdd`替换为`genSub`，将第12行的操作符替换为`OP_SUB`。

逻辑非运算和取负运算表达式的翻译比较简单，不过取负运算操作数限定为基本类型。

```
1  Var* GenIR::genMinus
   (Var*val){
2      if(!val->isBase()){
3          SEMERROR(EXPR_NOT_BASE);
4          return val;
5      }
6      Var*tmp=new Var(symtab.getScopePath(),KW_INT,false);
7      symtab.addVar(tmp);
8      symtab.addInst(new InterInst(OP_NEG
```

```
,tmp,val));  
9         return tmp;  
10 }
```

第2~5行处理操作数不是基本类型的语义错误。

第8行使用操作符OP_NEG产生取负运算表达式中间指令。

4. 后缀单目运算表达式

我们实现的后缀单目运算表达式只有两种：后缀自加和后缀自减运算表达式，因此实现比较简单。

```
1  Var* GenIR::genOneOpRight  
  (Var*val, Tag opt){  
2      if(!val)return NULL;  
3      if(val->isVoid()){  
4          SEMERROR(EXPR_IS_VOID);  
5          return NULL;  
6      }  
7      if(!val->getLeft()){  
8          SEMERROR(EXPR_NOT_LEFT_VAL);  
9          return val;  
10     }  
11     if(opt==INC)return genIncR  
    (val);                //右自加语句  
  
12     if(opt==DEC)return genDecR  
    (val);                //右自减语句  
  
13     return val;  
14 }
```

第3~6行处理操作数为void变量的情况。

第7~10行处理操作数不是左值的情况。

第11~12行进行后缀自加、自减运算表达式的翻译。

后缀自加、自减运算表达式翻译也需要指针运算的结果，后缀自加运算表达式的翻译如下：

```
1  Var* GenIR::genIncr
   (Var*val){
2      Var*tmp=genAssign
   (val);                                     //复制,
   tmp=val
3      if(val->isRef()){                       //( *p)++情况
   => t1=*p t2=t1+1 *p=t2
4      Var* t2=genAdd
   (tmp,Var::getStep(val));                   //t2=tmp+1
5      genAssign
   (val,t2);                                  //*p=t2
6      }
7      else
8          symtab.addInst(new InterInst(OP_ADD
9          ,val,val,Var::getStep(val)));
   //val=val+1
10     return tmp;
11 }
```

后缀自加运算表达式翻译与前缀自加运算表达式的翻译非常相似，只不过第2行对单参数genAssign函数的调用是无条件的，即必须做一次将val变量的值复制到tmp中，并且返回值是tmp而非val。这正是为了满足后缀自加运算是先返回操作数结果，再进行自加运算的特性。

后缀自减运算表达式翻译与后缀自加运算表达式翻译的形式相同，只需要将第4行的genAdd替换为genSub，将第8行的操作符

OP_ADD替换为OP_SUB。

5.数组索引运算表达式

对于数组索引运算表达式，我们将之转化为指针运算处理。例如数组索引运算表达式“a[i]”，转化为指针运算形式为“* (a+i)”。即首先执行数组名a与索引i的加法操作，结果保存到tmp。然后执行对tmp的指针运算操作。实现代码如下：

```
1  Var* GenIR::genArray
   (Var*array,Var*index){
2      if(!array || !index)return NULL;
3      if(array->isVoid()||index->isVoid()){
4          SEMERROR(EXPR_IS_VOID);
5          return NULL;
6      }
7      if(array->isBase() || !index->isBase()){
8          SEMERROR(ARR_TYPE_ERR);
9          return index;
10     }
11     return genPtr
   (genAdd
   (array,index));
12 }
```

第3~6行处理数组变量和索引变量为void类型的语义错误。

第7~10行表示数组变量为基本类型，或索引变量为非基本类型时，报告语义错误。

第11行产生数组索引运算表达式的中间代码，即首先调用genAdd函数执行数组与索引的加法，然后调用genPtr函数进行指针运算操

作。

6.函数调用表达式

函数调用表达式的翻译分为两个部分：传递参数和函数调用。

在设计中间代码时，操作符OP_ARG表示实际参数入栈操作。对每个实际参数，我们使用genPara函数生成入栈的中间代码指令。

```
1 void GenIR::genPara
  (Var*arg){
2     if(arg->isRef())arg=genAssign(arg);
3     InterInst*argInst=new InterInst(OP_ARG
    ,arg);
4     symtab.addInst(argInst);
5 }
```

第2行处理参数变量为指针运算结果的情况。

第3行生成OP_ARG中间代码指令，用于将arg压栈。

函数调用表达式翻译的实现代码如下：

```
1 Var* GenIR::genCall
  (Fun*function,vector<Var*>& args){
2     if(!function)return NULL;
3     for(int i=args.size()-1;i>=0;i--){           //逆向传递实际参数

4         genPara
  (args[i]);
5     }
6     if(function->getType()==Kw_VOID){
7         symtab.addInst(new InterInst(OP_PROC
    ,function));
```

```
8          return Var::getVoid(); //返回
void特殊变量

9      }
10     else{
11         Var*ret=new Var(symtab.getScopePath()
12             ,function->getType(),false);
13         symtab.addInst(new InterInst(OP_CALL
14             ,function,ret));
14         symtab.addVar(ret);
15         return ret;
16     }
17 }
```

第3~5行按照C语言参数从右向左的入栈规则生成参数入栈指令。

第6~9行处理无返回值（**void**）函数调用，生成**OP_PROC**函数调用指令，并返回**void**变量标识函数的返回类型。

第10~16行处理有返回值函数调用，生成**OP_CALL**函数调用指令，并返回函数调用结果变量**ret**。

3.5.5 复合语句与break、continue语句翻译

在高级语言程序中，如果说表达式是程序计算的抽象，那么语句便是程序控制的抽象，因为程序的控制逻辑通过语句来表达。我们将语句分为三类：分支语句、循环语句和break、continue语句。站在计算机机器语言角度，程序的执行无外乎两种形式：顺序和跳转。反映在程序计数器pc上便是取下一条指令执行，或者跳转到目标地址取指执行。一般的，不考虑使用goto语句的情况下，如果是顺着程序执行流的方向跳转，则表现为高级语言的分支语句，如果逆着程序执行流的方向跳转，则表现为高级语言的循环语句。因此，对语句的代码生成需要明确跳转指令和目标标签的位置。

1.if-else、switch-case分支语句

高级语言中最常用的分支语句应该是if-else语句，其基本形式为：

```
if(cond){  
    //do true  
}else{  
    //do false  
}
```

将其翻译为中间代码形式为：

```
//do cond  
OP_JF cond _else
```

```
//if(!cond)goto _else
```

```
//do true
OP_JMP _exit
```

```
//goto _exit_else:
```

```
//do false_exit:
```

如果if-else语句中只包含if部分，而没有else部分：

```
if(cond){
    //do true
}
```

则翻译为中间代码形式为：

```
//do cond
OP_JF cond _else
```

```
//if(!cond)goto _else
```

```
//do true_else:
```

注释的do cond部分处理条件表达式，do true和do false部分为if-else分支语句的内部实现代码，将其抽出剩下的便是if-else分支语句的实现框架，包括四个部分：

- 1) if首部：产生目标标签为_else的OP_JF指令。
- 2) else首部：产生目标标签为_exit的无条件跳转指令OP_JMP和标签_else。
- 3) else尾部：产生_exit标签。

4) if尾部: 当不存在else语句时, 产生_else标签。

对if-else分支语句的代码生成的实现如下:

```
1 void Parser::ifstat
2 {
3     symtab.enter();
4     InterInst* _else, *_exit;           // 标签
5
6     match(KW_IF);
7     if(!match(LPAREN))
8         recovery(EXPR_FIRST, LPAREN_LOST, LPAREN_WRONG);
9     Var* cond=expr();
10    ir.genIfHead
11    (cond, _else);                       // if头部
12
13    if(!match(RPAREN))
14        recovery(F(LBRACE), RPAREN_LOST, RPAREN_WRONG);
15    block();
16    symtab.leave();
17    if(F(KW_ELSE)){
18        //有
19        else
20            ir.genElseHead
21            (_else, _exit);               // else头部
22
23        elsestat();
24        ir.genElseTail
25        (_exit);                         // else尾部
26
27    }
28    else{
29        //无
30        else
31            ir.genIfTail
32            (_else);
33    }
34 }
35 void GenIR::genIfHead
36 (Var* cond, InterInst*& _else){
37     _else=new InterInst();              // 产生
38     else 标签
39
40     if(cond){
41         if(cond->isRef())cond=genAssign(cond);
42         symtab.addInst(new InterInst(OP_JF
```

```

    ,_else,cond));
27     }
28 }
29 void GenIR::genIfTail

(InterInst*& _else){
30     symtab.addInst(_else

);
31 }
32 void GenIR::genElseHead

(InterInst* _else,InterInst*& _exit){
33     _exit=new InterInst();           //产生
exit标签

34     symtab.addInst(new InterInst(OP_JMP
,_exit));
35     symtab.addInst(_else

);
36 }
37 void GenIR::genElseTail

(InterInst*& _exit){
38     symtab.addInst(_exit

);
39 }

```

代码第1~22行为ifstat递归下降子程序的实现代码，第23~39行为if首部、else首部、else尾部、if尾部的代码生成实现。

第8行在条件表达式计算完毕后，调用genIfHead处理if首部，在第26行生成目标标签为_else的OP_JF指令。

第18~20行在不存在else时，调用genIfTail处理if尾部，在第30行生成_else标签。

第14行调用genElseHead处理else首部，第34~35行生成OP_JMP指令目标标签为_exit，以及_else标签。

第16行调用genElseTail处理else尾部，第38行生成_exit标签。

标签对象的创建是使用无参构造函数InterInst，即调用genLb生成一个唯一的标签名称，将之保存在InterInst的label变量内。

```
1  string GenIR::genLb
   (){
2      lbNum++;
3      string lb="@L";
4      stringstream ss;
5      ss<<lbNum;
6      return lb+ss.str();
7  }
```

函数genLb的实现很简单，就是使用字符串“@L”后紧跟一个全局唯一的编号lbNum。

switch-case分支语句是另一种常见的分支语句，其基本形式为：

```
switch(cond){
    case lb_1:                                     //do
lb_1
    case lb_2:                                     //do
lb_2
    ...
    default:                                     //do
default
}
```

将其翻译为中间代码形式为：

```
//do cond
OP_JNE exit_1 lb_1 cond

                                     //if(lb_1!=cond)goto exit_1
//do lb_1exit_1:

    OP_JNE exit_2 lb_2 cond
//if(lb_2!=cond)goto exit_2
//do lb_2
```

```
exit_2:
    ...
    //do default_exit:
```

其中，标签_exit是switch-case的出口标签。实际case语句内不会产生到_exit的跳转，那么最终的default语句总是无条件执行。因此，常使用break语句强制退出一个case语句，即生成到_exit标签的跳转，比如：

```
switch(cond){
    case lb_1:                                     //do
lb_1        break;
    case lb_2:                                     //do
lb_2        break;
    ...
    default:                                       //do
default
}
}
```

将其翻译为中间代码形式为：

```
//do cond
OP_JNE exit_1 lb_1 condition

                                //if(lb_1!=cond)goto exit_1
//do lb_1
OP_JMP _exit                     //goto
_exitexit_1:

    OP_JNE exit_2 lb_2 condition    //if(lb_2!=cond)goto
exit_2                             //do lb_2
    OP_JMP _exit                     //goto
_exitexit_2:

    ...
    //do default_exit:
```

这样每一个case语句便可以独立执行了，当所有case语句都无法执行时，便执行default语句，这与C语言的switch-case语句的语义一致。同样的，我们将switch-case语句框架分为四个部分：

1) switch首部：本身不生成代码，但需要创建出口标签_exit指令对象，并保存_exit，为其内部可能出现的break语句的代码生成提供跳转标签信息。

2) case首部：生成跳转到_case_exit的条件跳转指令OP_JNE，条件比较操作数是case的常量标签lb_*和cond表达式的结果。

3) case尾部：生成case的退出标签_case_exit。

4) switch尾部：生成switch语句的退出标签_exit。

switch-case分支语句的代码生成的实现如下：

```
1 void Parser::switchstat
   (){
2     symtab.enter();
3     InterInst*_exit;
   // 标签

4     ir.genSwitchHead
   (_exit);                                     //switch头部

5     match(KW_SWITCH);
6     if(!match(LPAREN))
7         recovery(EXPR_FIRST, LPAREN_LOST, LPAREN_WRONG);
8     Var*cond=expr();
9     if(cond->isRef())cond=ir.genAssign(cond);
10    if(!match(RPAREN))
11        recovery(F(LBRACE), RPAREN_LOST, RPAREN_WRONG);
12    if(!match(LBRACE))
13        recovery(F(KW_CASE)_(KW_DEFAULT),
```



```

14             LBRACE_LOST, LBRACE_WRONG);
15     casestat(cond);
16     if(!match(RBRACE))
17         recovery(TYPE_FIRST||STATEMENT_FIRST,
18             RBRACE_LOST, RBRACE_WRONG);
19     ir.genSwitchTail

(_exit);                                     //switch尾部

20     symtab.leave();
21 }
22 void Parser::casestat

(Var*cond){
23     if(match(KW_CASE)){
24         InterInst*_case_exit;           //
    标签

25         Var*lb=caselabel();
26         ir.genCaseHead

(cond, lb, _case_exit);                     //case头部

27         if(!match(COLON))
28             recovery(TYPE_FIRST||STATEMENT_FIRST,
29                 COLON_LOST, COLON_WRONG);
30         symtab.enter();
31         subprogram();
32         symtab.leave();
33         ir.genCaseTail

(_case_exit);                             //case尾部

34         casestat(cond);
35     }
36     else if(match(KW_DEFAULT)){
//default默认执行

37         if(!match(COLON))
38             recovery(TYPE_FIRST||STATEMENT_FIRST,
39                 COLON_LOST, COLON_WRONG);
40         symtab.enter();
41         subprogram();
42         symtab.leave();
43     }
44 }
45 void GenIR::genSwitchHead

(InterInst*& _exit){
46     _exit=new InterInst();
    //产生

exit标签

47     push(NULL, _exit);
    //不允许

continue
48 }
49 void GenIR::genSwitchTail

```

```

(InterInst* _exit){
50      symtab.addInst(_exit

);                                     //添加

exit标签

51      pop();
52 }
53 void GenIR::genCaseHead

(Var*cond,Var*lb,
54      InterInst*& _case_exit){
55      _case_exit=new InterInst();    //产
生

case的

exit标签

56      if(lb)symtab.addInst(new InterInst(OP_JNE

/
57          _case_exit,cond,lb));
58 }
59 void GenIR::genCaseTail

(InterInst* _case_exit){
60      symtab.addInst(_case_exit

);                                     //添加

case的

exit标签

61 }

```

第1~21行为switchstat递归下降子程序的实现，第22~44行为casestat递归下降子程序的实现。第45~61行实现了switch首部、case首部、case尾部、switch尾部的代码生成。

第4行调用genSwitchHead处理switch语句首部。第46~47行创建_exit标签指令对象并使用push函数保存。push与pop函数与break、continue语句的翻译相关，后面会详细描述。

第19行调用genSwitchTail处理switch语句尾部，第50行生成_exit标签。

第26行调用genCaseHead处理case语句首部，第56~57行生成目标标签为_case_exit的OP_JNE指令。

第33行调用genCaseTail处理case语句尾部，第60行添加_case_exit标签。

2.while、do-while、for循环语句

while循环语句的基本形式为：

```
while(cond){  
    //do loop  
}
```

将其翻译为中间代码形式为：

```
_while:  
  
    //do cond  
    OP_JF cond _exit  
  
    //do loop  
    OP_JMP _while  
  
                                //if(!cond)goto _exit  
  
                                //goto _while_exit:
```

while循环语句的实现框架包括三个部分。

1) **while**首部: 创建循环入口标签`_while`和循环出口标签`_exit`的指令对象并保存, 为`break`和`continue`语句代码生成提供跳转标签信息。

2) **while**条件: 处理循环条件, 尤其是空循环条件。生成目标标签为`_exit`的`OP_JF`指令。循环条件表达式的计算必须在循环内部处理, 即在标签`_while`之后, 因为每次循环需要重新计算循环表达式的值。

3) **while**尾部: 产生目标标签为`_while`的`OP_JMP`指令, 并添加标签`_exit`。

对**while**循环语句的代码生成的实现如下:

```
1 void Parser::whilestat
2 {
3     sytab.enter();
4     InterInst* _while, *_exit;           //标签

5     ir.genWhileHead
6     (_while, _exit);                     //while循环头部

7     match(KW_WHILE);
8     if(!match(LPAREN))
9         recovery(EXPR_FIRST || F(RPAREN),
10                 LPAREN_LOST, LPAREN_WRONG);
11     Var* cond = altexpr();
12     ir.genWhileCond
13     (cond, _exit);                       //while条件

14     if(!match(RPAREN))
15         recovery(F(LBRACE), RPAREN_LOST, RPAREN_WRONG);
16     block();
17     ir.genWhileTail
18     (_while, _exit);                     //while尾部

19     sytab.leave();
20 }
21 void GenIR::genWhileHead
```

```

(InterInst*& _while,
18      InterInst*& _exit){
19      _while=new InterInst();           //产生

while 标签

20      symtab.addInst(_while
);                                     //添加

while 标签

21      _exit=new InterInst();           //产生

exit 标签

22      push(_while,_exit);             //进
入

while
23 }
24 void GenIR::genWhileCond
(Var*cond,InterInst* _exit){
25     if(cond){
26         if(cond->isVoid())cond=Var::getTrue();    //处理空表达式

27         else if(cond->isRef())cond=genAssign(cond);
28         symtab.addInst(new InterInst(OP_JF
,_exit,cond));
29     }
30 }
31 void GenIR::genWhileTail
(InterInst*& _while,
32     InterInst*& _exit){
33     symtab.addInst(new InterInst(OP_JMP
,_while));
34     symtab.addInst(_exit
);                                     //添加

exit 标签

35     pop();
//离开

while
36 }

```

代码第1~16行为whilestat递归下降子程序的实现代码，第11~36行为while首部、while条件、while尾部的代码生成实现。

第4行调用genWhileHead处理while循环首部。第19~21创建_while和_exit标签指令对象。并添加_while标签。

第10行在while循环表达式处理完毕后，调用genWhileCond处理循环条件。第26行当cond为void变量则调用getTrue，返回常量1。第28行根据循环条件生成目标标签为_exit的OP_JF指令。

第14行调用genWhileTail处理while循环尾部。第33~34行生成OP_JMP指令，添加_exit标签。

do-while循环语句的基本形式为：

```
do{  
    //do loop  
}while(cond);
```

将其翻译为中间代码形式为：

```
_do:  
  
    //do loop  
    //do cond  
    OP_JF cond _do  
  
                                //if(cond)goto _do_exit:
```

do-while循环语句的实现框架包括两个部分。

1) **do-while**首部: 创建循环入口标签`_do`和循环出口标签`_exit`的指令对象, 并保存, 为`break`和`continue`语句代码生成提供条件标签信息, 并添加`_do`标签。

2) **do-while**尾部: 处理循环条件, 尤其是空循环条件。产生目标标签为`_do`的`OP_JT`指令, 并添加标签`_exit`。循环条件表达式的计算必须在循环内部处理, 因为每次循环需要重新计算循环表达式的值。

对**do-while**循环语句的代码生成的实现如下:

```
1 void Parser::dowhilestat
() {
2     syntab.enter();
3     InterInst* _do, * _exit;                                // 标签

4     ir.genDowhileHead
(_do, _exit);                                                // do-while 头部

5     match(KW_DO);
6     block();
7     if (!match(KW_WHILE))
8         recovery(F(LPAREN), WHILE_LOST, WHILE_WRONG);
9     if (!match(LPAREN))
10        recovery(EXPR_FIRST | F(RPAREN),
11                 LPAREN_LOST, LPAREN_WRONG);
12    syntab.leave();
13    Var* cond = altexpr();
14    if (!match(RPAREN))
15        recovery(F(SEMICON), RPAREN_LOST, RPAREN_WRONG);
16    if (!match(SEMICON))
17        recovery(TYPE_FIRST | STATEMENT_FIRST | F(RBRACE),
18                 SEMICON_LOST, SEMICON_WRONG);
19    ir.genDowhileTail
(cond, _do, _exit);                                          // do-while 尾部

20 }
21 void GenIR::genDowhileHead
(InterInst*& _do,
22  InterInst*& _exit) {
23    _do = new InterInst();                                    // 产生
do 标签
```

```

24      _exit=new InterInst();                                //产生
exit标签

25      symtab.addInst(_do
);
26      push(_do,_exit);                                    //进入
do-while
27 }
28 void GenIR::genDoWhileTail
(Var*cond,InterInst* _do,
29      InterInst* _exit){
30      if(cond){
31          if(cond->isVoid())cond=Var::getTrue();
32          else if(cond->isRef())cond=genAssign(cond);
33          symtab.addInst(new InterInst(OP_JT
,_do,cond));
34      }
35      symtab.addInst(_exit
);
36      pop();
37 }

```

代码第1~27行为dowhilestat递归下降子程序的实现代码，第28~37行为do-while首部、do-while尾部的代码生成实现。

第4行调用genDoWhileHead处理do-while循环首部。第23~25行创建_do和_exit标签指令对象，并使用push记录。添加_do标签。

第19行在do-while循环表达式处理完毕后，调用genDoWhileTail处理循环尾部。第31行当cond为void变量则调用getTrue返回常量1，第33行根据循环条件生成目标标签为_do的OP_JT指令。第35行添加循环出口标签_exit。

for循环的代码翻译比前两者复杂，for循环语句的基本形式为：

```
for(init;cond;step){//do  
loop  
}
```

其中init为初始化语句部分，cond为循环条件语句，step为循环因子控制语句。将其翻译为中间代码形式为：

```
    //do init  
_for:  
    //do cond  
    OP_JF cond _exit//if(!cond)goto  
_exit  
    //do loop  
  
    //do step  
  
    OP_JMP _for//goto _for  
_exit:
```

for循环的init语句仅执行一次，因此在循环体外即_for标签前计算。而cond和step必须在循环体内计算，因此在_for标签后。

虽然上述的for循环翻译方式是最佳的，其中对do loop的处理在do step之前。但实际扫描源代码的顺序是step在loop之前，如果按照边扫描边生成代码的要求，是无法生成上述中间代码的。当然，我们可以选择先生成step的代码并缓存，等loop代码生成后将缓存的代码追加到loop之后。不过，我们选择了“偷懒”的翻译方式，这样做在一定程度上降低了for循环的代码性能。

```
    //do init_for:  
  
    //do cond  
    OP_JF _exit cond
```

```
        OP_JMP _block
    _step:

        //do step
        OP_JMP _for
    _block:

        //do loop
        OP_JMP _step
    _exit:
```

for循环语句的实现框架包括四个部分。

- 1) for首部: 创建循环入口标签_for和循环出口标签_exit指令对象并保存, 为break和continue语句代码生成提供条件标签信息并添加标签_for。
- 2) for条件首部: 处理循环条件, 尤其是空循环条件。创建循环体入口标签_block、循环因子控制语句入口标签_step指令对象。生成目标标签为_exit的OP_JF指令。生成目标标签为_block的OP_JMP指令以跳转到循环体。添加循环控制语句入口标签_step。
- 3) for条件尾部: 处理完step语句后, 添加目标标签为_for的OP_JMP指令以跳转到循环开始位置。添加循环体入口标签_block。
- 4) for尾部: 产生目标标签为_step的OP_JMP指令, 并添加标签_exit。

对for循环语句的代码生成的实现如下:

```
1 void Parser::forstat
   (){
2     symtab.enter();
3     InterInst *_for,*_exit,*_step,*_block;
4     match(KW_FOR);
5     if(!match(LPAREN))
6         recovery(TYPE_FIRST|EXPR_FIRST|F(SEMICON),
7                 LPAREN_LOST,LPAREN_WRONG);
8     forinit();
9     ir.genForHead
   (_for,_exit);
10    Var*cond=altexpr();
11    ir.genForCondBegin
   (cond,_step,_block,_exit);
12    if(!match(SEMICON))
13        recovery(EXPR_FIRST,SEMICON_LOST,SEMICON_WRONG);
14    altexpr();
15    if(!match(RPAREN))
16        recovery(F(LBRACE),RPAREN_LOST,RPAREN_WRONG);
17    ir.genForCondEnd
   (_for,_block);
18    block();
19    ir.genForTail
   (_step,_exit);
20    symtab.leave();
21 }
22 void GenIR::genForHead
   (InterInst*& _for,InterInst*& _exit){
23     _for=new InterInst();
24     _exit=new InterInst();
25     symtab.addInst(_for
   );
26 }
27 void GenIR::genForCondBegin
   (Var*cond,InterInst*& _step,
28    InterInst*& _block,InterInst* _exit){
29     _block=new InterInst();
30     _step=new InterInst();
31     if(cond){
32         if(cond->isVoid())cond=Var::getTrue();
33         else if(cond->isRef())cond=genAssign(cond);
34         symtab.addInst(new InterInst(OP_JF
   ,_exit,cond));
35         symtab.addInst(new InterInst(OP_JMP
   ,_block));
36     }
37     symtab.addInst(_step
   );
38     push(_step,_exit);
```

```

39 }
40 void GenIR::genForCondEnd

(InterInst* _for, InterInst* _block){
41     symtab.addInst(new InterInst(OP_JMP
, _for));
42     symtab.addInst(_block

);
43 }
44 void GenIR::genForTail

(InterInst*& _step, InterInst*& _exit){
45     symtab.addInst(new InterInst(OP_JMP
, _step));
46     symtab.addInst(_exit

);
47     pop();
48 }

```

代码第1~21行为forstat递归下降子程序的实现代码，第22~48行为for首部、for条件首部、for条件尾部、for尾部的代码生成实现。

第9行调用genForHead处理for循环首部。第23~25行创建循环入口标签_for和循环出口标签_exit指令对象并保存，并添加标签_for。

第11行在for循环表达式处理完毕后，调用genForCondBegin处理循环条件首部。第32行当cond为void变量则调用getTrue，返回常量1。第34~35行根据循环条件生成目标标签为_exit的OP_JF指令，以及目标标签为_block的OP_JMP指令。

第17行调用genForCondEnd处理循环条件尾部。第41~42行生成目标标签为_for的OP_JMP指令，并添加循环体入口标签_block。

第19行调用genForTail处理for循环尾部，第45~46行生成目标标签为_step的OP_JMP指令，并添加标签_exit。

3.break 、 continue语句

在C语言语法中，**break**语句用于中断循环语句或者**case**语句，而**continue**语句用于中断本次循环。从代码生成角度看，**break**语句是产生到循环（或**switch**）语句出口的无条件跳转，而**continue**语句则是产生到循环语句入口的无条件跳转。例如：

```
while(cond){
    //do something
    break;

    continue;

    //do something
}
```

翻译为中间代码形式为:

```

_while:
    //do cond
    OP_JF _exit cond                //if(!cond)goto
_exit
    //do something
    OP_JMP _exit

                                    //break

    OP_JMP _while

                                    //continue

    //do something
    OP_JMP _while                //goto _while
_exit:

```

在翻译**break**或**continue**语句时，必须获得正确的跳转指令的目标标签。 **break**或**continue**语句只能出现在循环语句或者**switch-case**语句内部，因此在对循环和**switch-case**语句进行代码生成时，必须产生入口和出口标签。

另外，循环语句和**switch-case**语句允许相互嵌套，而**break**和**continue**语句的作用对象是最内层的循环或**switch-case**语句。这与变量的作用域管理机制比较相似，因此可以采用类似作用域管理的方法。

```
1  vector<InterInst*> heads
;
2  vector<InterInst*> tails
;
3  void GenIR::push
   (InterInst*head, InterInst*tail){
4      heads.push_back(head);
5      tails.push_back(tail);
6  }
7  void GenIR::pop
   (){
8      heads.pop_back();
9      tails.pop_back();
10 }
```

我们使用两个栈**heads**和**tails**对循环语句和**switch-case**语句的嵌套层次进行动态管理，其中**heads**记录语句的入口标签序列，**tails**记录语句的出口标签序列。这两个序列是同时操作的，当进入循环语句或**switch**语句作用域时，使用**push**函数将语句的入口标签和出口标签分别保存到**heads**和**tails**栈中。当离开循环语句或**switch**语句作用域时，使用**pop**函数将**heads**和**tails**的栈顶元素弹出。回顾前面讨论的循环语句和

switch-case语句的代码生成，在处理语句的首部和尾部的代码生成时，会调用push和pop函数完成出入口标签的动态管理。这样heads和tails的栈顶元素始终保存着当前作用域所在循环或switch-case语句的入口和出口标签，如果当前作用域不在任何循环或switch语句内，栈顶元素应该保存NULL。因此，需要对heads和tails进行初始化。

```
push(NULL, NULL
);                                     //初始化
```

对break和continue语句进行代码生成时，只需要从heads和tails栈顶取出入口和出口标签即可。

```
1 void GenIR::genBreak
(){
2     InterInst*tail=tails.back
();                                     //取出口标签

3     if(tail)symtab.addInst(new InterInst(OP_JMP
,tail));
4     else SEMERROR
(BREAK_ERR);                         //break不在循环或
switch-case中

5 }
6 void GenIR::genContinue
(){
7     InterInst*head=heads.back
();                                     //取入口标签

8     if(head)symtab.addInst(new InterInst(OP_JMP
,head));
9     else SEMERROR
```

```

        (CONTINUE_ERR);                                //continue不在循环中

10 }

```

第2~4行，使用genBreak进行break语句代码生成时，需要从tails栈中取出语句出口标签tail，然后生成目标标签为tail的OP_JMP指令。如果tail为NULL，则说明break语句不在合法的语句内，报告语义错误。

第7~9行，使用genContinue进行continue语句代码生成时，需要从heads栈中取出语句入口标签head，然后生成目标标签为head的OP_JMP指令。如果head为NULL，则说明continue语句不在合法的语句内，报告语义错误。

在前面讨论的循环语句和switch-case语句的代码生成中，产生的语句入口和出口标签分别如表3-9所示。

表3-9 语句出入口标签

语句类型	入口标签	出口标签
while 循环	<code>_while</code>	<code>_exit</code>
do-while 循环	<code>_do</code>	<code>_exit</code>
for 循环	<code>_for</code>	<code>_exit</code>
switch-case 分支	NULL	<code>_exit</code>
其他	不处理	不处理

其中最为特殊的是switch-case语句的入口标签，它的入口标签设置为NULL。这是因为switch-case语句内不允许出现continue语句，因此在

switch-case语句内进行continue语句的代码生成时，取出的heads栈顶元素为NULL，因此报告语义错误。

3.5.6 目标代码生成

经过前面的讨论，我们已经将高级语言代码转化为中间代码形式。接下来要将中间代码翻译为具体的目标指令集指令，我们选择生成Intel x86指令集指令。设计中间代码除了可以屏蔽具体机器的指令集细节，还可以降低代码生成时需要考虑的变量存储访问的复杂性。在中间代码表示中，对变量的信息统一由Var对象管理，这使得中间代码指令形式简单。而将中间代码进一步转化为目标指令集代码时，则需要考虑变量的存储细节。

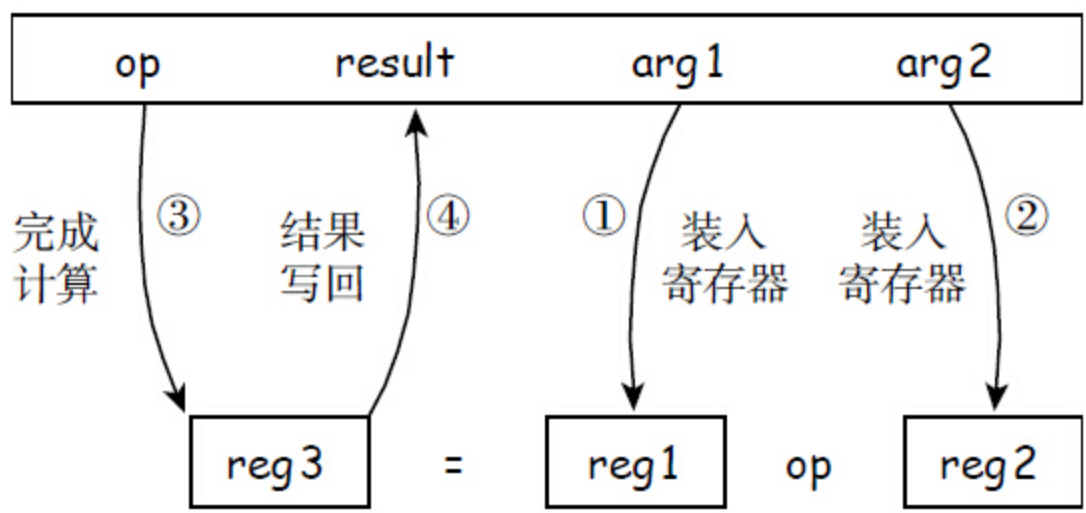


图3-25 目标代码生成

如图3-25所示，我们根据中间代码指令的一般形式讨论目标代码的生成策略。中间代码指令（四元式）分为四个基本要素：`arg1`和`arg2`

提供计算对象，`op`提供计算操作，`result`提供计算结果。由于`arg1`、`arg2`和`result`在中间代码指令`InterInst`对象内由`Var`对象统一表示，因此可能存在多种存储类型——常量、数据段、栈、堆、寄存器等，变量存储的多样性使得生成的目标指令中操作数的组合有数十种之多。为了简化目标代码生成的烦琐程度，我们使用寄存器代替原本存储形式多样的操作数，并将中间代码指令的目标代码生成划分为四个阶段：

- 1) 加载`arg1`到寄存器`reg1`，此时需要根据`arg1`的存储类型取出`arg1`的值，并保存到寄存器`reg1`。
- 2) 加载`arg2`到寄存器`reg2`，此时需要根据`arg2`的存储类型取出`arg2`的值，并保存到寄存器`reg2`。
- 3) 生成计算指令，根据`op`的特性选择适当的目标指令操作码，并以`reg1`和`reg2`为操作数进行计算，计算结果保存到`reg3`。在x86指令集中，`reg1`和`reg3`一般是同一个寄存器。比如指令“`add eax, ebx`”的功能便是“`eax=eax+ebx`”，其中`reg1`和`reg3`都是寄存器`eax`。
- 4) 计算结果写回，此时需要考虑`result`的存储类型，将`reg3`的值写回到`result`。

在讨论计算指令的生成前，需要明确变量是如何加载和写回的。`loadVar`函数将变量加载到寄存器。

```

1 void InterInst::loadVar
(string reg32,string reg8,Var*var){
2     if(!var)return;
3     const char*reg=var->isChar()?reg8.c_str():reg32.c_str();
4     if(var->isChar())
5         emit("mov %s,0",reg32.c_str());           //字符
,将
32位寄存器清
0
6     const char*name=var->getName().c_str();
7     if(var->notConst()){
8         int off=var->getOffset();
9         if(!off){
//全局符号

10         if(!var->getArray())
//mov eax/al,[name]
11             emit("mov %s,[%s]",reg,name);
12         else
//mov eax,name
13             emit("mov %s,%s",reg,name);
14         }
15     }else{
//局部符号

16         if(!var->getArray())
//mov eax/al,[ebp-off]
17             emit("mov %s,[ebp%+d]",reg,off);
18         else
//lea eax,[ebp-off]
19             emit("lea %s,[ebp%+d]",reg,off);
20         }
21     }
22 }else{
//常量

23         if(var->isBase())                                //mov
eax,val
24             emit("mov %s,%d",reg,var->getVal());
25         else
//mov eax,name
26             emit("mov %s,%s",reg,name);
27     }
28 }

```

第3行判断`var`如果是字符变量，则将变量值保存到寄存器的低8位，否则用寄存器的全32位保存变量值。第4~5行，在`var`是字符变量的时候，将32位寄存器清0。

第7~21行处理var不是常量的情况。第10~11行表示var是全局变量，直接根据变量名访问内存，如“mov eax, [var]”。第12~13行表示var是全局数组，对数组名的访问不是取内存的值，而是将数组名作为立即数，如“mov eax, array”。第16~17行表示var是局部变量，需要根据变量的栈帧偏移访问内存，如“mov eax, [ebp-40]”。第12~13行表示var是局部数组，局部数组名的访问一般使用lea指令取数组第一个元素的地址，如“lea eax, [ebp-4]”。

第22~27行处理var是常量的情况。第23~24行表示var是基本类型常量，使用var的getVal方法将常量值取出作为立即数，如“mov eax, 100”。第25~26行表示var是字符串常量，由于字符串常量在数据段，因此与全局数组名的访问相同，即使用字符串常量名称作为立即数，如“mov eax, str”。

loadVar函数是将变量的值取出保存到寄存器，但是在取址运算的时候，对变量的访问不是取值，而是取址。因此需要使用leaVar函数实现变量地址的加载。

```
1 void InterInst::leaVar
  (string reg32, Var*var){
2     if(!var)return;
3     const char*reg=reg32.c_str();
4     const char*name=var->getName().c_str();
5     int off=var->getOffset();
6     if(!off)
//mov eax,name
7         emit("mov %s,%s",reg,name);
8     else
//lea eax,[ebp-off]
9         emit("lea %s,[ebp%+d]",reg,off);
10 }
```

变量的地址加载比较简单，数组变量和常量是不需要取地址的，不需要考虑，而普通的变量就只有全局和局部之分。第6~7行表示取全局变量的地址，只需要将变量名作为立即数访问即可，如“`mov eax, var`”。第8~9行表示取局部变量的地址，这个与局部数组名的访问类似，即使用`lea`指令获取变量地址，如“`mov eax, [ebp-4]`”。

讨论完变量的加载后，接下来讨论计算结果的写回。我们使用`storeVar`函数将寄存器的值写回到变量。

```
1 void InterInst::storeVar
  (string reg32,string reg8,Var*var){
2     if(!var)return;
3     const char*reg=var->isChar()?reg8.c_str():reg32.c_str();
4     const char*name=var->getName().c_str();
5     int off=var->getOffset();
6     if(!off)                                     //mov
[name],eax/al
7         emit("mov [%s],%s",name,reg);
8     else
//mov [ebp-off],eax/al
9         emit("mov [ebp+%d],%s",off,reg);
10 }
```

第3行判断`var`如果是字符变量，则选择将寄存器低八位的值保存到变量，否则将32位寄存器的值保存到变量。

就像对变量取址一样，计算结果的写回也只需要考虑普通变量的情况。第6~7行表示写入全局变量，将变量名作为地址访问，如“`mov[var], eax`”。第8~9行表示写入局部变量，需要根据局部变量的栈帧偏移访问内存，如“`mov[ebp-4], eax`”。

除了将计算结果写回到变量之外，局部变量的初始化也需要对变量进行写操作。

```
1 void InterInst::initVar
  (Var*var){
2     if(!var)return;
3     if(!var->unInit()){
4         if(var->isBase())                //mov
    eax, val
5         emit("mov eax,%d",var->getVal());
6     else
    //mov [name],ptrVal
7         emit("mov eax,%s",var->getPtrVal().c_str());
8         storeVar("eax","al",var);
9     }
10 }
```

第3行判断变量是否被初始化，`unInit`函数检测`var`的`inited`字段是否为`false`，只有使用常量初始化的变量，其`inited`字段才设为`true`（详见3.4.1节`setInit`函数实现）。第4~5行表示基本类型变量的初始化，只需要调用`getVal`函数将变量的初始值取出，作为立即数保存到`eax`寄存器，如“`mov eax, 100`”。

第6~7行表示字符指针变量的初始化，其初始值必是常量字符串。通过调用`getPtrVal`函数获取字符串常量的名称，并作为立即数保存到`eax`寄存器，如“`mov eax, str`”。

第8行调用`storeVar`将`eax`寄存器的值写入变量`var`，完成变量的初始化。

目标代码生成中，获取操作数的值或者地址只需要调用loadVar和leaVar将需要的操作数加载到寄存器，写入计算结果时只需要调用storeVar将寄存器写入内存即可，这使得计算指令的翻译大大简化。函数toX86将中间代码指令转化为x86汇编指令。

```
1  #define emit
   (fmt, args...) fprintf(file, "\t" fmt "\n", ##args)
2  void InterInst::toX86
   () {
3      if(label!=""){
4          fprintf(file, "%s:\n", label.c_str());
5          return;
6      }
7      switch(op){
8          case OP_DEC:
9              initVar(arg1);
10             break;
11          case OP_ENTRY:
12              emit("push ebp");
13              emit("mov ebp, esp");
14              emit("sub esp, %d", inst->getFun()->getMaxDep());
15              break;
16          case OP_EXIT:
17              emit("mov esp, ebp");
18              emit("pop ebp");
19              emit("ret");
20              break;
21          case OP_AS:
22              loadVar("eax", "a1", arg1);
23              storeVar("eax", "a1", result);
24              break;
25          case OP_ADD:
26              loadVar("eax", "a1", arg1);
27              loadVar("ebx", "b1", arg2);
28              emit("add eax, ebx");
29              storeVar("eax", "a1", result);
30              break;
31          case OP_SUB:
32              loadVar("eax", "a1", arg1);
33              loadVar("ebx", "b1", arg2);
34              emit("sub eax, ebx");
35              storeVar("eax", "a1", result);
36              break;
37          case OP_MUL:
38              loadVar("eax", "a1", arg1);
39              loadVar("ebx", "b1", arg2);
40              emit("imul ebx");
41              storeVar("eax", "a1", result);
42              break;
43          case OP_DIV:
44              loadVar("eax", "a1", arg1);
45              loadVar("ebx", "b1", arg2);
46              emit("idiv ebx");
47              storeVar("eax", "a1", result);
```



```

48         break;
49     case OP_MOD:
50         loadVar("eax", "a1", arg1);
51         loadVar("ebx", "b1", arg2);
52         emit("idiv ebx");
53         storeVar("edx", "d1", result);
54         break;
55     case OP_NEG:
56         loadVar("eax", "a1", arg1);
57         emit("neg eax");
58         storeVar("eax", "a1", result);
59         break;
60     case OP_GT:
61         loadVar("eax", "a1", arg1);
62         loadVar("ebx", "b1", arg2);
63         emit("mov ecx, 0");
64         emit("cmp eax, ebx");
65         emit("setg cl");
66         storeVar("ecx", "c1", result);
67         break;
68     case OP_GE:
69         loadVar("eax", "a1", arg1);
70         loadVar("ebx", "b1", arg2);
71         emit("mov ecx, 0");
72         emit("cmp eax, ebx");
73         emit("setge cl");
74         storeVar("ecx", "c1", result);
75         break;
76     case OP_LT:
77         loadVar("eax", "a1", arg1);
78         loadVar("ebx", "b1", arg2);
79         emit("mov ecx, 0");
80         emit("cmp eax, ebx");
81         emit("setl cl");
82         storeVar("ecx", "c1", result);
83         break;
84     case OP_LE:
85         loadVar("eax", "a1", arg1);
86         loadVar("ebx", "b1", arg2);
87         emit("mov ecx, 0");
88         emit("cmp eax, ebx");
89         emit("setle cl");
90         storeVar("ecx", "c1", result);
91         break;
92     case OP_EQU:
93         loadVar("eax", "a1", arg1);
94         loadVar("ebx", "b1", arg2);
95         emit("mov ecx, 0");
96         emit("cmp eax, ebx");
97         emit("sete cl");
98         storeVar("ecx", "c1", result);
99         break;
100    case OP_NE:
101        loadVar("eax", "a1", arg1);
102        loadVar("ebx", "b1", arg2);
103        emit("mov ecx, 0");
104        emit("cmp eax, ebx");
105        emit("setne cl");
106        storeVar("ecx", "c1", result);
107        break;
108    case OP_NOT:
109        loadVar("eax", "a1", arg1);
110        emit("mov ebx, 0");
111        emit("cmp eax, 0");
112        emit("sete b1");
113        storeVar("ebx", "b1", result);

```

```

114         break;
115     case OP_AND:
116         loadVar("eax", "a1", arg1);
117         emit("cmp eax, 0");
118         emit("setne cl");
119         loadVar("ebx", "b1", arg2);
120         emit("cmp ebx, 0");
121         emit("setne bl");
122         emit("add eax, ebx");
123         storeVar("eax", "a1", result);
124         break;
125     case OP_OR:
126         loadVar("eax", "a1", arg1);
127         emit("cmp eax, 0");
128         emit("setne al");
129         loadVar("ebx", "b1", arg2);
130         emit("cmp ebx, 0");
131         emit("setne bl");
132         emit("or eax, ebx");
133         storeVar("eax", "a1", result);
134         break;
135     case OP_JMP:
136         emit("jmp %s", target->label.c_str());
137         break;
138     case OP_JT:
139         loadVar("eax", "a1", arg1);
140         emit("cmp eax, 0");
141         emit("jne %s", target->label.c_str());
142         break;
143     case OP_JF:
144         loadVar("eax", "a1", arg1);
145         emit("cmp eax, 0");
146         emit("je %s", target->label.c_str());
147         break;
148     case OP_JNE:
149         loadVar("eax", "a1", arg1);
150         loadVar("ebx", "b1", arg2);
151         emit("cmp eax, ebx");
152         emit("jne %s", target->label.c_str());
153         break;
154     case OP_ARG:
155         loadVar("eax", "a1", arg1);
156         emit("push eax");
157         break;
158     case OP_PROC:
159         emit("call %s", fun->getName().c_str());
160         emit("add esp, %d", fun->getParaVar().size()*4);
161         break;
162     case OP_CALL:
163         emit("call %s", fun->getName().c_str());
164         emit("add esp, %d", fun->getParaVar().size()*4);
165         storeVar("eax", "a1", result);
166         break;
167     case OP_RET:
168         emit("jmp %s", target->label.c_str());
169         break;
170     case OP_RETV:
171         loadVar("eax", "a1", arg1);
172         emit("jmp %s", target->label.c_str());
173         break;
174     case OP_LEA:
175         leaVar("eax", arg1);
176         storeVar("eax", "a1", result);
177         break;
178     case OP_SET:
179         loadVar("eax", "a1", result);

```

```

180             loadVar("ebx", "b1", arg1);
181             emit("mov [ebx], eax");
182             break;
183         case OP_GET:
184             loadVar("eax", "a1", arg1);
185             emit("mov eax, [eax]");
186             storeVar("eax", "a1", result);
187             break;
188     }
189 }

```

第1行的emit宏是对fprintf的封装，表示将一条指令写入文件，并在指令前添加一个制表符，在指令结尾处添加换行符。

第3行label字段不为空，表示输出标签指令，在标签后添加字符‘:’和换行符。第7~187行处理所有的中间代码指令的翻译。

第8~10行处理OP_DEC指令，调用initVar处理变量的初始化。

第11~15行处理OP_ENTRY指令，输出函数入口代码。包括ebp入栈、保存esp、开辟栈帧。第16~20行处理OP_EXIT指令，输出函数出口代码。包括恢复esp、恢复ebp、函数返回。

第21~24行处理OP_AS指令，调用loadVar将arg1加载到寄存器eax（或8位寄存器al），再调用storeVar将寄存器eax写入变量result。

第25~54行处理双目算术运算指令加、减、乘、除、取模，对应操作符为OP_ADD、OP_SUB、OP_MUL、OP_DIV、OP_MOD。它们都是调用loadVar将arg1和arg2加载到eax和ebx（或8位寄存器bl），然后生成运算指令，分别为“add eax, ebx”、“sub eax, ebx”、“mul

ebx”、“div ebx”、“div ebx”，最后调用storeVar将eax写入result。对于取模运算，则是将edx（或8位寄存器dl）写入result。

第55~59行处理取负单目算术运算指令，操作符为OP_NEG。调用loadVar将arg1加载到寄存器eax，生成计算指令“neg eax”，再调用storeVar将寄存器eax写入变量result。

第60~107行处理关系运算指令，对应操作符为OP_GT、OP_GE、OP_LT、OP_LE、OP_EQU、OP_NE。首先调用loadVar将arg1和arg2加载到eax和ebx，然后将比较结果保存到ecx，生成的指令序列如下：

```
mov ecx,0
cmp eax,ebx
set? cl
```

其中，“set?”指令会根据cmp指令的比较结果将ecx设置为1。上述关系运算对应的“set?”指令分别为“setgt”、“setge”、“setlt”、“setle”、“sete”、“setne”。最后调用storeVar将ecx（或8位寄存器cl）写入result。

第108~114行处理逻辑非运算指令，操作符为OP_NOT。调用loadVar将arg1加载到寄存器eax，然后生成如下指令序列，将结果保存到ebx。

```
mov ebx,0
cmp eax,0
sete bl
```

首先将ebx设为0，然后比较eax是否等于0，eax如果等于0则将ebx设为1，这样ebx保存了eax的逻辑非结果。最后调用storeVar将寄存器ebx写入变量result。

第115~134行处理逻辑与、逻辑或运算指令，操作符分别为OP_AND、OP_OR。首先调用loadVar将arg1加载到寄存器eax，然后生成指令序列“cmp eax, 0”和“setne al”，如果eax不等于0，则设置eax为1，eax保存了arg1的逻辑值。按照类似的方式处理arg2，即调用loadVar将arg2加载到寄存器ebx，然后生成指令序列“cmp ebx, 0”和“setne bl”，这样ebx保存了arg2的逻辑值。接着生成运算指令“add eax, ebx”或“or eax, ebx”将运算结果存放到eax中，最后调用storeVar将寄存器eax写入变量result。

第135~137行处理无条件跳转指令OP_JMP，直接生成jmp指令，目标标签地址为target的label字段名。

第138~147行处理条件跳转指令OP_JT、OP_JF。首先将arg1加载到eax，然后生成“cmp eax, 0”指令，最后生成jne和je指令，目标标签地址为target的label字段名。

第148~153行处理条件跳转指令OP_JNE。首先将arg1加载到eax，将arg2加载到ebx，然后生成“cmp eax, ebx”指令，最后生成jne指令，目标标签地址为target的label字段名。

第154~157行处理参数入栈指令OP_ARG。首先将arg1加载到eax，然后使用“push eax”指令将参数入栈。

第158~166行处理函数调用指令OP_PROC和OP_CALL。首先使用call指令调用函数，函数名为fun的name字段。函数调用完毕后需要恢复栈帧，生成“add esp, len”指令，其中len为函数参数个数，即“getParaVar () .size () *4”（参数都是通过push eax入栈，因此每个参数占4个字节）。对于OP_CALL指令，还需要调用storeVar将函数返回值eax写入变量result。

第167~173行处理函数返回指令OP_RET和OP_RETV。对于OP_RETV指令需要调用loadVar将函数返回值arg1保存到eax。然后OP_RET和OP_RETV都需要使用jmp指令跳转到函数出口代码位置，即函数返回点returnPoint记录的标签指向的位置。

第174~177行处理取址运算指令OP_LEA，首先调用leaVar将arg1的地址保存到eax，然后调用storeVar将eax保存到result。

第178~187行处理指针运算指令OP_SET和OP_GET。

OP_SET指令的含义为“*arg1=result”，因此首先调用loadVar将result保存到eax，将arg1保存到ebx，最后生成“mov[ebx], eax”完成OP_SET运算。

OP_GET指令的含义为“result=*arg1”，因此首先调用loadVar将arg1保存到eax，然后生成“mov eax, [eax]”完成指针的取值，最后调用storeVar将eax保存到result。

3.5.7 数据段生成

在Linux系统中，将代码的静态数据放在3个独立的段内。

(1) “.data”段。该段保存所有已初始化的全局变量。

(2) “.bss”段。该段保存所有未初始化的全局变量，且初始化为0。

(3) “.rodata”段。该段保存所有常量字符串的内容。

为了减少段的数量，达到清晰说明编译系统实现的目的，我们将上述三个段合并到“.data”，统称为数据段。数据段的信息来源于全局变量的定义和常量字符串。

在符号表中，**varTab**保存了所有变量的信息，数据段只关心全局变量的定义。因此，需要提供从**varTab**中获取全局变量的方法。

```
1  vector<Var*> SymTab::getGlbVars
   (){
2      vector<Var*> glbVars;
3      hash_map<string, vector<Var*>*, string_hash>::iterator varIt
4          , varEnd=varTab.end();
5      for(varIt=varTab.begin(); varIt!=varEnd; ++varIt){
6          string varName=varIt->first;
7          if(varName[0]!='<')continue;                                //忽略常量

8          vector<Var*>&list=*varIt->second;
9          for(int j=0; j<list.size(); j++){
10             if(list[j]->getPath().size()==1){                        //全局变量
```



```
11                                     glbVars.push_back(list[j]);
12                                     break;
13                                     //唯一同名全局变量
14                                     }
15                                     }
16     return glbVars;
17 }
```

函数getGlbVars返回全局变量列表，第5行遍历varTab。

第6行取出变量名varName，如果以字符'<'开始，说明是数字常量，则忽略。

第8~9行取出同名变量列表list，并遍历。

第10行判断列表内每个变量的作用域路径是否长度为1，即作用域路径为“/0”。判断成功后表示该变量是全局变量，将变量对象添加到列表glbVars，并停止列表查找，因为全局作用域内不可能出现另一个同名的变量。

在符号表中，strTab保存了所有字符串常量的信息，数据段需要保存常量字符串的内容。词法分析器对字符串扫描后，将之转化为字符串的二进制表示，比如字符串“abc\n”在字符串常量的变量对象内，保存的字符串内容为'a'、'b'、'c'、'\n'。代码生成需要将字符串内容输出到汇编代码文件内，上述字符串输出后，换行符会按照字符格式打印，在文件内产生换行，而非输出“\n”。当然可以选择将特殊的字符

再次转化为转义字符输出，比如对于换行符，输出字符串'\n'。不过我们选择输出与NASM汇编语法相似的格式。

```
"abc",10,0
```

对于普通的字符串，我们输出字符串本身的内容并在字符串首尾加双引号。而对于特殊字符则将其转化为对应的ASCII码后输出，并且以逗号进行分隔。需要考虑的特殊字符有：制表符'\t'、换行符'\n'、双引号'"'、字符串结束符'\0'。

字符串转换的思想是，逐字节扫描字符串内容，依次处理每个字符的输出格式。对于特殊字符，输出其ASCII码，否则正常输出字符。我们使用变量chpass记录上一个字符的输出形式，0表示输出ASCII码，1表示输出字符。当上一个字符的输出形式是ASCII码时，如果当前字符输出形式仍是ASCII码，则需要插入逗号后再输出ASCII码，否则先后输出逗号和双引号（字符串开始），再输出字符。当上一个字符的输出形式是字符时，如果当前字符输出形式是ASCII码，则需要先后输出双引号（字符串结束）和逗号，再输出ASCII码，否则直接输出字符。另外，如果当前输出的字符是第一个字符时，则不需要插入逗号。如果当前输出的字符是最后一个字符，且输出形式不是ASCII码时，需要输出一个双引号表示字符串结束。所有字符处理完毕后，还需要输出一个逗号和0，表示结束标记。

将字符串转化为NASM语法规则的实现代码为：

```
1  string Var::getRawStr
2  (){
3      stringstream ss;
4      int len=strVal.size();
5      for(int i=0,chpass=0;i<len;i++){
6          if(strVal[i]==10
7              ||strVal[i]==9
8              ||strVal[i]=='\"
9              ||strVal[i]=='\0'){          //\n \t "
10
11              if(chpass==0)
12              {
13                  if(i!=0)ss<<",";
14                  ss<<(int)strVal[i];
15              }
16              else
17                  ss<<"\", "<<(int)strVal[i];
18              chpass=0;
19          }
20          else{
21              if(chpass==0){
22                  if(i!=0)ss<<",";
23                  ss<<"\"<<strVal[i];
24              }
25              else
26                  ss<<strVal[i];
27              if(i==len-1)ss<<"\"";
28              chpass=1;
29          }
30      }
31      ss<<",";
32      return ss.str();
33  }
```

第4行扫描字符串内的字符，第5~17行处理特殊字符的输出，第18~27行处理普通字符的输出，第29行处理字符串结束标记。

第11~13行处理输出ASCII码后仍输出ASCII码的情况，如果当前字符不是第一个字符则需要插入逗号。

第15行处理输出普通字符后输出ASCII码的情况，此时需要插入双引号和逗号。

第19~22行处理输出ASCII码后输出普通字符的情况，如果当前字符不是第一个字符则需要插入逗号，然后插入双引号。

第24行处理输出普通字符后输出普通字符的情况，此时直接将字符输出。

第25行判断输出的普通字符是否是最后一个字符，如果是则输出双引号。

第29行输出字符串结束标记，即逗号和数字0。

通过这样的转换，字符串“\thello\nworld!”被转化为如下形式：

```
9, "hello", 10, "world", 0
```

其中9为制表符的ASCII码，10为换行符的ASCII码，0为字符串结束标记。

在描述数据段生成之前，需要了解NASM汇编的数据定义语法：

```
<label> [times] <len> <value>
```

其中，

- 1) **label**部分表示任意的合法标识符，一般是变量名。
- 2) **times**可选部分表示后面数据的重复次数，比如“**times 100**”表示重复100次，一般用于定义数组。
- 3) **len**部分指单位内存大小，“**db**”表示一个字节、“**dw**”表示两个字节、“**dd**”表示四个字节。
- 4) **value**部分表示初始值，初始值可以是整数常量，可以是标识符，也可以是上述NASM格式的字符串。

例如以下全局变量定义。

```
char ch;                                     //变量未初始化，初始值为0
int var=100;                                //变量已初始化
int array[255];                             //全局数组的初始值为0
char*str="hello";                           //假设字符串常量为"hello"
@LO"
```

使用NASM的数据定义语法表示为：

```
ch db 0
//ch,
1字节，初值
0
```

```

var dd 100
//var,

4字节, 初值

100
array times 255 dd 0                                //array,

255×4字节, 初值

0
str dd @L0
//str,

4字节, 初值

@L0
@L0 db "hello",0                                //@L0,

6字节, 初值“
hello”

```

我们发现NASM语法定义的数据长度实际是times、len和value三者长度的乘积。

数据段生成实现代码如下：

```

1 void SymTab::genData
2 {
3     vector<Var*> glbVars=getGlbVars
4     ();
5     //全局变量
6
7     for(unsigned int i=0;i<glbVars.size();i++){
8         Var*var=glbVars[i];
9         fprintf(file,"global %s\n",var->getName().c_str());
10        fprintf(file,"\t%s ",var->getName().c_str());
11        int typeSize=var->getType()==KW_CHAR?1:4;
12        if(var->getArray())
13            //times 100
14            fprintf(file,"times %d ",var->getSize()/typeSize);
15        const char* type=var->getType()==KW_CHAR&&!var->getPtr()
16            ?"db":"dd";
17        fprintf(file,"%s ",type);
18        //db
19        dd
20        if(!var->unInit()){
21            //初始值
22
23
24            if(var->isBase())
25                //基本类型

```

```

15                                fprintf(file, "%d\n", var->getVal());
16                                else
// 字符指针

17                                fprintf(file, "%s\n", var->getPtrVal().c_str());
18                                }
19                                else
// 未初始化

20                                fprintf(file, "0\n");
21                                }
22                                hash_map<string, Var*, string_hash>::iterator strIt,
23                                strEnd=strTab.end();
// 常量字符串

24                                for(strIt=strTab.begin(); strIt!=strEnd; ++strIt){
25                                    Var* str=strIt->second;
26                                    fprintf(file,
27                                        "\t%s db %s\n",
28                                        str->getName().c_str(),
29                                        str->getRawStr

().c_str());                                //str db "abc",0
30                                }
31 }

```

第2~21行处理全局变量的翻译，第22~30行处理常量字符串的翻译。

第5~6行使用global声明变量名为全局符号，并输出变量名作为label部分。第7~9行处理数组，输出times部分，第10~12行处理内存单位大小，输出len部分，第13~21行处理变量的初值，输出value部分。

第14~15行处理基本类型变量的初值，调用getVal，输出变量的值。

第17行处理字符指针变量的初值，调用getPtrVal，输出字符串常量的名称。

第20行处理未初始化的变量，输出初值0。

第26~29行输出常量字符串的名称、db和字符串的NASM格式字符串内容。

经过目标代码生成和数据段生成，源代码被翻译为NASM格式的x86汇编指令程序，接下来将代码段和数据段整合，输出到汇编文件。

```
1 void SymTab::genAsm
2 (
3 {
4     fprintf(file, "section .data\n");           //数据段
5
6     genData
7
8 ();
9     fprintf(file, "section .text\n");           //代
10    码段
11
12     hash_map<string, Fun*, string_hash>::iterator funIt,
13         funEnd=funTab.end();
14     for(funIt=funTab.begin(); funIt!=funEnd; ++funIt){
15         Fun* fun=funIt->second;
16         fprintf(file, "global %s\n", fun->getName().c_str());
17         fprintf(file, "%s:\n", fun->getName().c_str());
18         vector<InterInst*>&code=fun->getInterCode();
19         vector<InterInst*>::iterator instIt, instEnd=code.end();
20         for(instIt=code.begin(); instIt!=instEnd; ++instIt){
21             (*instIt)->toX86
22
23 ();
24
25 }
26 }
27 }
28 }
```

第3~4行输出“section.data”声明数据段，并调用genData输出数据段内容。

第5行输出“`section.text`”声明代码段，第8~15行遍历函数表`funTab`输出每个函数的代码。

第10~11行使用`global`声明函数名为全局符号（我们认为函数是全局可见的），输出函数名，并以冒号结束，表示函数起始地址。第12行获取函数的中间代码。

第13~16行遍历函数的中间代码，并调用`toX86`将中间代码转化为x86汇编代码。

3.6 本章小结

本章我们根据已设计的编译器结构，分别从词法分析、语法分析、符号表管理、语义分析和代码生成的角度描述了一个简单的编译器实现。从大量的实例代码中，可以发现编译器实现的每一个细节。另外从实现编译器的过程中不仅能解开高级语言层面的很多疑惑，还能加深对计算机程序工作机制的理解，这对理解计算机工作原理的本质是非常重要的。

按照本章描述的编译器实现，我们发现生成的汇编代码繁琐而冗长，有些代码甚至是没有必要存在的。在第4章，我们将阐述如何使用优化技术使编译器生成的目标代码更简洁、高效。

第4章 编译优化

如切如磋，如琢如磨。

——《诗经》

没有编译优化功能的编译器生成的代码存在大量的冗余，无论是生成的中间代码还是生成的目标汇编代码。

例如，在中间代码生成的过程中，对于源代码表达式

```
a=1+2+3;
```

生成的中间代码形式为（为了更清晰地表达中间代码指令的含义，我们将四元式“<op, result, arg1, arg2>”表示为“result=arg1 op arg2”的形式，其中操作符op选用常用的运算符代替）：

```
t1=1+2  
t2=t1+3  
a=t2
```

我们发现表达式“a=1+2+3;”的实际效果是“a=6;”，而且在编译时期，完全可以计算出临时变量t1=3、t2=6。我们希望通过中间代码优化后，可以将上述三条中间代码指令缩减为一条。

```
a=6
```

这样的中间代码优化方式称为常量传播，即将变量的常量初值传递到使用变量的表达式中，尽可能计算出表达式的值，并依次将变量的值传播下去，达到简化代码的目的。除了常量传播，后面还会介绍复写传播、死代码消除等优化算法的实现。

在目标代码生成过程中，也存在冗余的情况，如中间代码指令：

```
a=b+c
```

生成的目标汇编代码形式为（假设变量a、b、c都是全局int类型变量）：

```
1  mov  eax,[a]
2  mov  ebx,[b]
3  add  eax,ebx
4  mov  [c],eax
```

我们发现第2、3条汇编指令可以用一条指令代替，上述汇编代码被转化为如下形式：

```
1  mov  eax,[a]
2  add  eax,[b]
3  mov  [c],eax
```

这样的目标代码优化方式称为窥孔优化，即通过发现指令模式，使用单一的指令代替多条功能等价的指令，从而达到简化代码的目的。

在第2章图2-9中描述的现代编译器结构中，将编译器分为前端、优化器和后端三个部分。编译器的后端包含了指令选择、寄存器分配和指令调度，本质上它们与代码的优化息息相关。我们着重描述寄存器分配的实现，对指令选择和指令调度不做说明，对此感兴趣的读者可以参考其他编译器相关资料进行学习。

基于以上的讨论，我们对优化器的设计如图4-1所示。

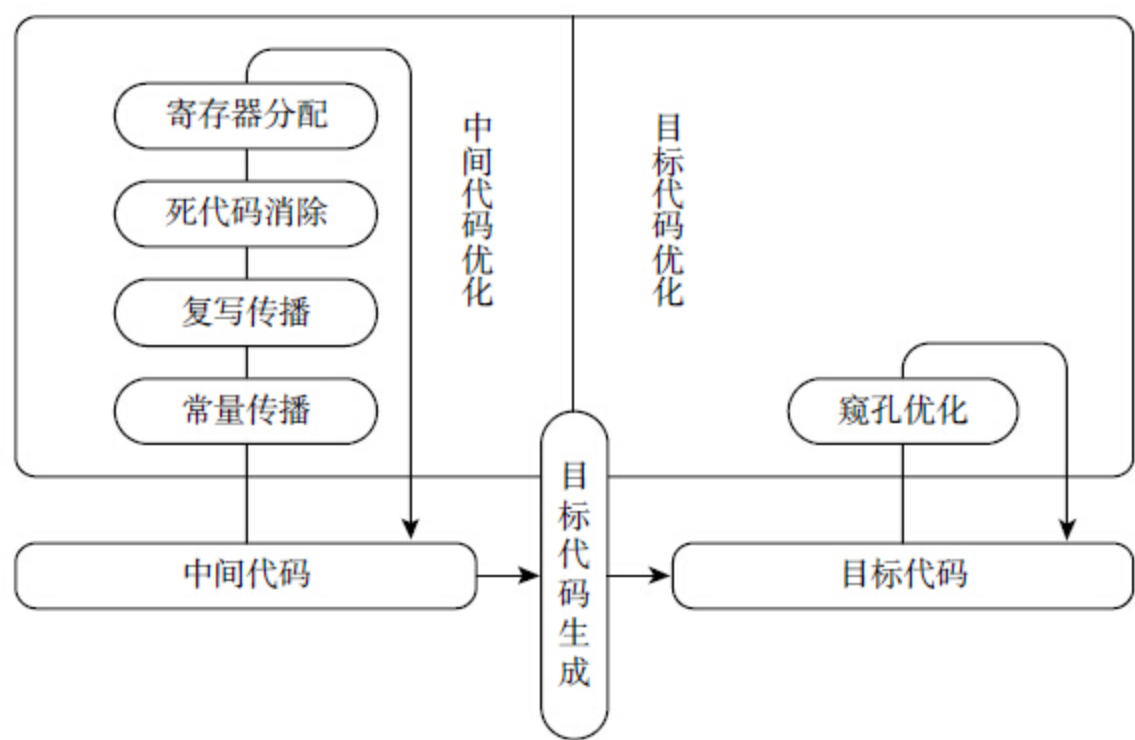


图4-1 优化器结构

根据3.5节中的描述，中间代码经过目标代码生成被翻译为目标汇编代码。因此，编译优化分为中间代码优化和目标代码优化两个部分。如图4-1所示，在我们实现的优化器中，中间代码优化经过常量传

播、复写传播和死代码消除三个过程。寄存器分配可以选择在目标代码生成之前进行，即为中间代码分配寄存器。目标代码优化由窥孔优化实现。

4.1 数据流分析

中间代码优化一般是在数据流分析的基础上进行的，且满足通用的数据流分析框架。

如图4-2所示，中间代码优化首先为中间代码构造流图（控制流图），然后对流图进行数据流分析，并获得需要的数据流信息，最后根据数据流信息指导中间代码的优化。数据流分析处理的对象是流图，因此在进行数据流分析之前，需要了解流图的构造。

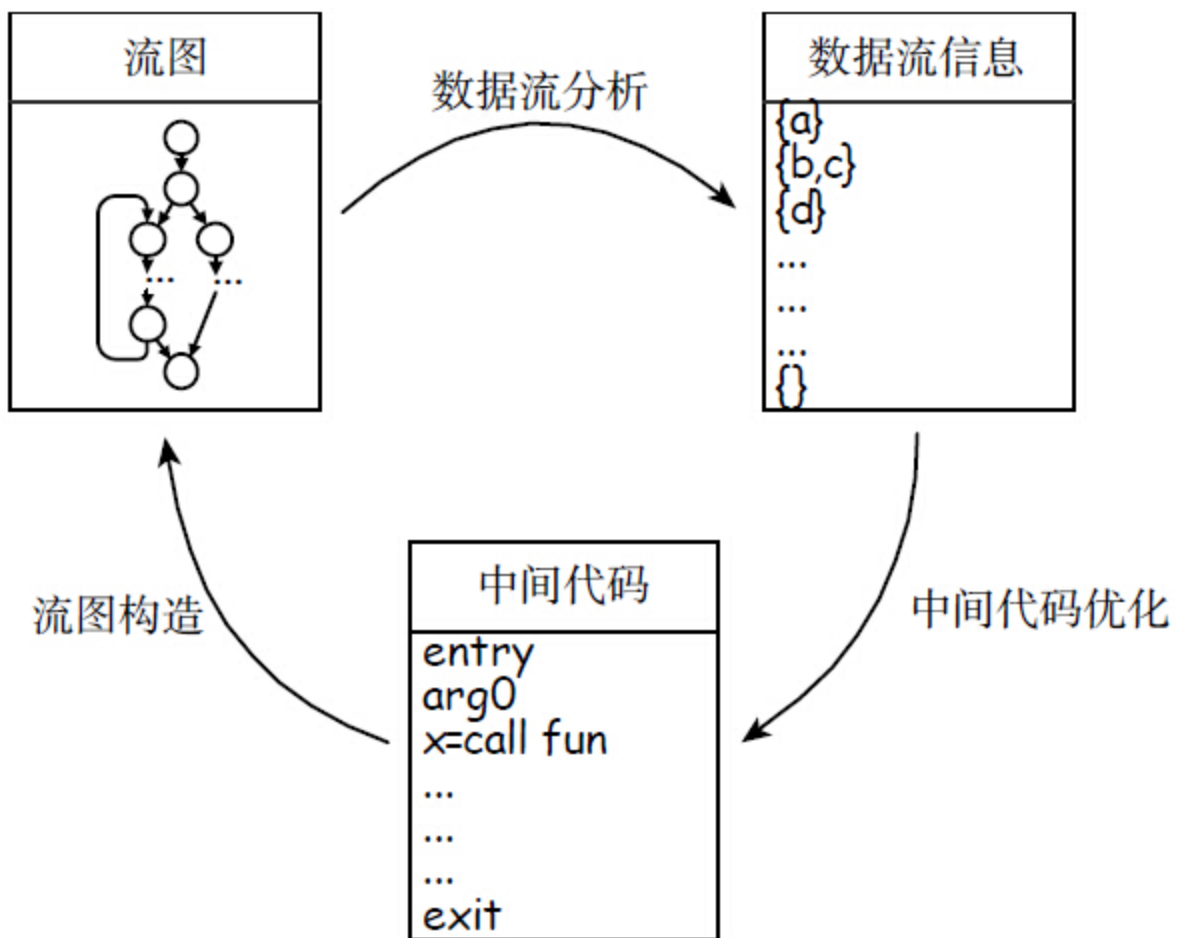


图4-2 基于数据流分析的中间代码优化

4.1.1 流图

流图是有向有环图，满足图数据结构的基本性质。一般包含一个入口节点和一个出口节点，流图的走向总是从入口节点到出口节点，且流图中允许出现环。流图的节点称为基本块，流图的边表示基本块间的跳转关系。

为了构造流图，需要了解首指令的概念。中间代码的首指令定义如下：

- 1) 第一条指令。
- 2) 跳转指令的目标指令。
- 3) 紧跟跳转指令之后的指令。

中间代码首指令和下条首指令前的所有指令组成的整体称为基本块。根据首指令的定义可以确定，基本块内的指令是顺序执行的，不存在跳转分支。

如图4-3所示，函数f的实现代码被翻译为中间代码。在中间代码中，根据首指令的定义，指令0是第一条指令，指令4和指令10是跳转指令的目标指令，指令6紧跟跳转指令5之后，因此指令0、4、6、10是首指令。根据基本块的定义，流图共包含4个基本块，对应的指令编号

序列分别为“指令0~3”、“指令4~5”、“指令6~9”和“指令10”。为了保证流图仅包含一个入口和一个出口，在流图开始处添加入口基本块Entry，在流图结束处添加出口基本块Exit。入口和出口基本块恰好与中间代码指令OP_ENTRY和OP_EXIT对应，因此将这两条指令分别作为独立的基本块。

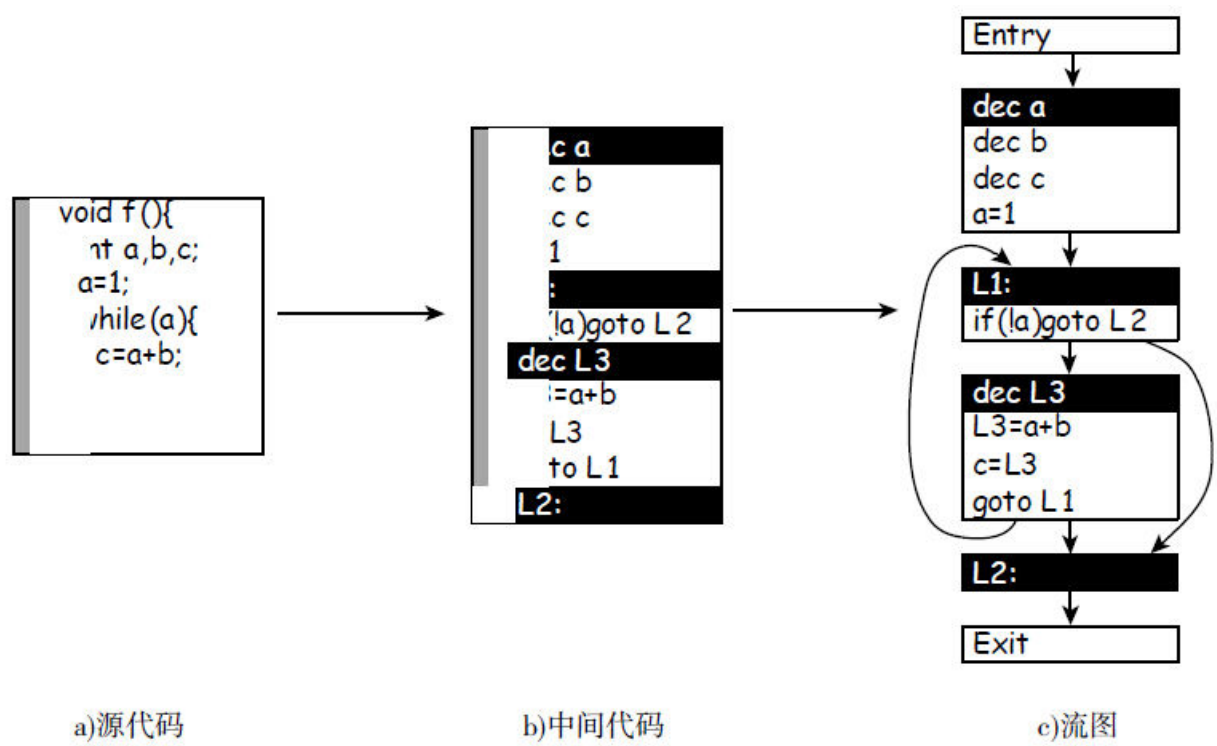


图4-3 流图构造

接下来讨论流图构造的实现，首先是首指令的标识。

```
1 void InterCode::markFirst
() {
2     unsigned int len=code.size();           //最少两条指令
3     code[0]->setFirst();
//OP_ENTRY
```

```

4      code[len-1]->setFirst();
//OP_EXIT
5      code[1]->setFirst();                                     //第
一条

6      for(unsigned int i=1;i<len-1;++i){
7          if(code[i]->isJmp()||code[i]->isJcond()){
8              code[i]->getTarget()-
setFirst();                                     //跳转

9              code[i+1]->setFirst();
//紧跟跳转

10         }
11     }
12 }

```

变量`code`是`vector<InterInst*>`类型，记录了函数所有的中间代码。在中间代码生成时，无论函数体是否为空，总是生成函数入口指令`OP_ENTRY`和函数出口指令`OP_EXIT`。因此，`code`内至少包含两条指令。

由于需要构造基本块`Entry`和`Exit`，因此第3~4行将`code`的第一条和最后一条指令标识为首指令。`setFirst`函数设置了指令`InterInst`对象的`first`字段。第5行的`code[1]`是函数内真正的第一条指令，满足首指令的条件1，因此需要标识为首指令。

第6~11行处理首指令的条件2和3，第7行判断指令`code[i]`是否是跳转指令，`isJmp`表示无条件跳转指令，`isJcond`表示条件跳转指令。第8行将跳转指令的目标指令标识为首指令，第9行将跳转指令后紧跟的指令标识为首指令。

完成中间代码code的首指令标记后，便可以基于首指令生成流图。流图构造涉及的关键数据结构和字段如下：

```
1  class Block
2  {
3      public:
4          list<InterInst*> insts;                //指令序列
5
6          list<Block*>prevs;                      //前驱
7
8          list<Block*>succs;                      //后继
9
10 };
11 class DFG
12 {
13     public:
14         vector<InterInst*> codeList;            //中间代码
15
16         vector<Block*>blocks;                  //所有基本块
17
18 };
19
```

其中Block类表示基本块，DFG类表示流图。DFG的codeList字段记录构造流图的中间代码，blocks字段记录所有的基本块。Block的insts字段记录基本块内所有的指令序列，prevs字段记录基本块的所有前驱，即直接到达该基本块的所有基本块，succs字段记录基本块的所有后继，即该基本块直接到达的所有基本块。基于这样的设计，我们分两步实现流图的构造：先根据首指令构造基本块对象，再确定基本块对象间的前驱和后继关系。

基本块对象的构造实现如下：

```

1 void DFG::createBlocks
2 {
3     vector<InterInst*>tmpList; //临时列表
4
5     tmpList.push_back(codeList[0]); //第一条指令
6
7     for(unsigned int i=1;i<codeList.size();++i){
8         if(codeList[i]->isFirst()){
9             blocks.push_back(new Block(tmpList)); //添加基本块
10
11             tmpList.clear(); //清除临时列表
12
13         }
14         tmpList.push_back(codeList[i]); //添加指令
15     }
16
17     blocks.push_back(new Block(tmpList)); //最后的基本块
18 }

```

构造基本块时，使用**tmpList**缓存每个基本块内的指令序列。

第3行将第一条指令（OP_ENTRY）添加到**tmpList**中，该指令是首指令。

第4~10行处理后继的指令。第5行判断指令如果是首指令，则根据**tmpList**内缓存的指令创建基本块，然后清空**tmpList**。第9行将新的首指令或后继指令添加到**tmpList**。

第11行添加最后一个基本块Exit（仅包含指令OP_EXIT）。

根据**tmpList**创建基本块的流程为：

```

1  Block::Block
   (vector<InterInst*>&codes){
2      for(unsigned int i=0;i<codes.size();++i){
3          codes[i]->block=this;                //记录指令所在的基本块

4          insts.push_back(codes[i]);           //转换为

list
5      }
6  }

```

Block的insts字段是list<InterInst*>类型，而tmpList是vector<InterInst*>类型，因此需要逐个添加指令到insts中。除此之外，将指令的block字段设置为当前基本块，以便于通过指令直接访问基本块。

确定基本块前驱和后继关系的过程称为连接基本块，实现如下：

```

1  void DFG::linkBlocks
   (){
2      for (unsigned int i = 0; i < blocks.size(); ++i){
3          InterInst*last=blocks[i]->insts.back();        //基本
块最后指令

4          if(last->isJmp()||last->isJcond()){              //
跳转

5          Block*tar=last->last->getTarget()->
block;                //目标基本块

6          blocks[i]->succs.push_back(tar);
//后继

7          tar->prevs.push_back(blocks[i]);
//前驱

8      }
9      if(!last->isJmp()&&i!=block.size()-1){              //顺序

10         blocks[i]->succs.push_back(blocks[i+1]);
//后
继

```

```
11          blocks[i+1]->prevs.push_back(blocks[i]);          //前  
12          }  
13      }  
14 }
```

第2行遍历所有的基本块，依次处理。第3行取出基本块的最后一条指令`last`。

第4行判断`last`如果是跳转指令，则取出`last`的目标指令所在的基本块`tar`，并更新当前基本块`blocks[i]`的后继和`tar`的前驱。

第7行判断`last`如果不是直接跳转指令，且不是最后一个基本块（最后一个基本块不会有后继），则更新当前基本块`blocks[i]`的后继和紧接着下个基本块`blocks[i+1]`的前驱。

经过`createBlocks`和`linkBlocks`的处理，DFG内保存了流图的完整信息。

4.1.2 数据流分析框架

基于流图的数据流分析是面向问题的，不同的问题要求，其数据流分析的实现也不尽相同。例如，对于图4-3的流图，假定需要根据数据流分析统计函数执行时最少执行的中间代码指令数（包含标签指令）。

如图4-4所示，统计最少执行指令个数需要按照代码的执行顺序进行计算，而且只关注基本块出口处（out）计算的指令个数。

初始化阶段，每个基本块的out都需要一个初值。对于Entry入口块，其初值为1，表示已经执行了OP_ENTRY指令。对于其他基本块，初值为最大正整数（inf），这是因为需要计算最少指令个数。

初始化完毕后，依次更新每个基本块（除了Entry块）的out值，每次处理完所有基本块称为一遍处理。每一遍处理时，总是按照如下原则进行计算：

$$B.in = \min (\text{所有} B \text{的前驱}.out)$$
$$B.out = B.in + B \text{的指令数}$$

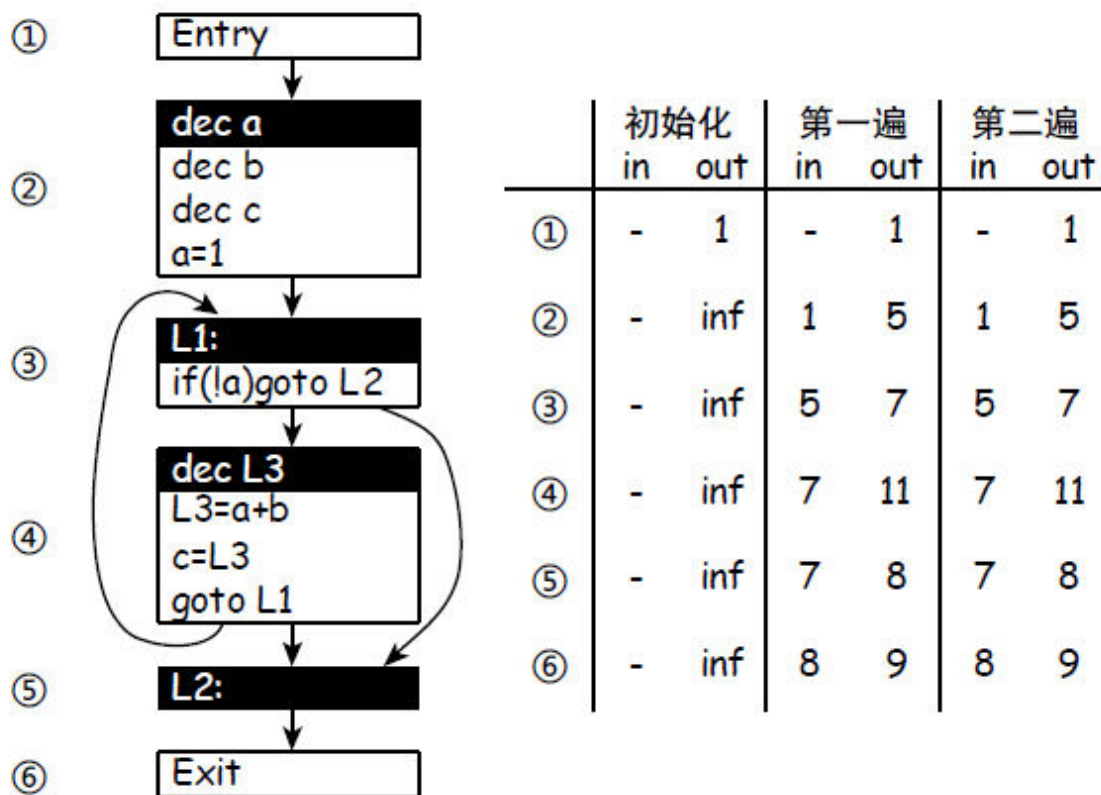


图4-4 最少指令数数据流分析

例如第一遍处理时，基本块2只有一个前驱基本块0，因此其in值为基本块0的out值1。基本块2包含4条指令，因此基本块2的out值为其in值加上基本块2的指令数，结果为5。而对于基本块3，其有两个前驱基本块2和4，基本块2和4的out值分别为5和inf，取最小值为5。基本块3包含2条指令，因此基本块3的out值为其in值加上基本块3的指令数，结果为7。按照这样的方式依次计算其他基本块的值，基本块4的out值为11，基本块5的out值为8，Exit块的out值为9。

由于第一遍处理后，存在基本块的out值被更新的情况，需要第二遍处理。而第二遍处理后，所有基本块的out保持不变，因此终结数据

流分析的过程。这样不断地循环进行多遍处理，直到运算结果保持稳定时终止计算的过程，称为迭代不动点计算。最终，Exit基本块的out值便是最少执行指令的个数，即9条，从数据流图中不难看出这一点，执行路径经过的基本块为1、2、3、5、6。

从上述例子中，可以看出数据流分析需要考虑如下基本要素：

1) 数据流方向。数据分析过程是与代码执行顺序相同（称为前向）还是相反（称为逆向）。

2) 值集。包含初值和数据流分析中使用的值。就初值而言，对于前向数据流，即Entry块的out初值和其他基本块的out初值。对于逆向数据流，则是Exit的in值和其他基本块的in值。一般称前者为边界集合，后者为初值集合。

3) 交汇函数。对于前向数据流，在计算基本块in值时，如果基本块有多个前驱，则需要提供一种计算方式将前驱的out“合并”。对于逆向数据流，在计算基本块out值时，如果基本块有多个后继，则需要提供一种计算方式将后继的in“合并”。

4) 传递函数。对于前向数据流，需要提供一种计算方式将基本块的in值转化为基本块out值。对于前向数据流，则需要提供一种计算方式将基本块的out值转化为基本块的in值。

以上数据流分析的基本要素是所有数据流分析所共有的，一般称为数据流分析框架。数据流分析框架是数据流分析问题的抽象，与具体问题无关。不同问题的数据流分析只是上述基本要素不同。

对于数据流分析框架，其形式化定义为一个四元组 (D, V, Λ, F) 。其中 D 为数据流分析的方向，分为前向和逆向。 V 为半格的值集，为数据流信息的全集。 Λ 为半格的交汇运算，用于合并不同路径的数据流信息。 F 为数据流图基本块的传递函数集合，定义了数据流信息经过基本块后的变化规则。通常使用数据流方程表示一个具体的数据流问题，前向和逆向数据流方程的基本形式如下：

$$\begin{array}{cc} \text{前向} & \left\{ \begin{array}{l} \text{Entry.out} = v_{\text{Entry}} \\ \text{B.out} = T \\ \text{B.in} = \bigwedge_{p \in \text{prec}(\text{B})} (p.\text{out}) \\ \text{B.out} = f_{\text{B}}(\text{B.in}) \end{array} \right. & \text{逆向} & \left\{ \begin{array}{l} \text{Exit.in} = v_{\text{Exit}} \\ \text{B.in} = T \\ \text{B.out} = \bigwedge_{s \in \text{succ}(\text{B})} (s.\text{in}) \\ \text{B.in} = f_{\text{B}}(\text{B.out}) \end{array} \right. \end{array}$$

其中 **Entry** 表示入口基本块，**Exit** 表示出口基本块，**B** 表示一般的基本块。 v 表示边界集合， T 表示初值集合，它们都是半格值集 V 的元素。 Λ 符号表示交汇运算， f_{B} 表示基本块 **B** 的传递函数。对于一个具体的数据流问题，只需要确定数据流框架四元组 (D, V, Λ, F) 的值，其中值集 V 包含边界集合 v 和初值集合 T 。例如上述求最少执行指令数的问题，其数据流方程为：

$$\text{前向} \left\{ \begin{array}{l} \text{Entry.out} = 1 \\ \text{B.out} = \text{inf} \\ \text{B.in} = \min_{p \in \text{prec}(\text{B})} (p.\text{out}) \\ \text{B.out} = \text{B.in} + \text{B.num} \end{array} \right.$$

其中B.num表示基本块B的指令个数。由此可以确定数据流框架的基本要素的值：D为前向、V为正整数集合（边界值v为1，初值T为最大正整数inf）、Λ为最小值运算min、f_B 为计算公式 B.out=B.in+B.num。

根据数据流方程，很容易实现数据流分析。前向数据流分析实现的伪代码如下：

```
Entry.out=vEntry
//初始化

Entry.out
for(B!=Entry)B.out=T //初始化

B.out
while(B.out changed) //迭代不动点计算

    for(B!=Entry){ //遍历基本块
        B
            B.in=Λ
        p∈prec(B)
        (p.out) //计算
        B.in
            B.out=fB
        (B.in) //计算
        B.out
    }
```

类似地，逆向数据流分析实现的伪代码如下：

```
Exit.in=vExit
//初始化

Exit.in
for(B!=Exit)B.in=T //初始化

B.in
```

```

while(B.in changed)                                     //迭代不
动点计算

    for(B!=Exit){                                       //遍历基
本块
        B
            B.out=1
        S∈succ(B)
        (s.in)                                           //计算
        B.out
            B.in=fB
        (B.out)                                           //计算
        B.in
    }

```

上述讨论的求最少执行指令数的数据流分析实现的伪代码如下：

```

Entry.out=1                                             //初始化
Entry.out
for(B!=Entry)B.out=-1                                   //初始化
B.out
while(B.out changed)                                   //迭代不动点计算

    for(B!=Entry){                                       //遍历基本块
        B
            B.in=minp∈prec(B)
        (p.out)                                           //计算
        B.in
            B.out=B.in+B.num                               //计算
        B.out
    }

```

最终计算结果保存在Exit.out内。

4.2 中间代码优化

中间代码优化算法一般是基于数据流分析框架实现的。与前面讨论的数据流问题示例不同的是，实际编译系统中的中间代码优化算法的数据流框架的基本要素内容更多样，数据流问题描述更为复杂。我们使用常量传播、复写传播和变量活跃性的数据流分析，完成对应的中间代码优化。

4.2.1 常量传播

常量传播利用编译时可以确定的变量值代替变量，提前进行表达式求值，消除不必要的运算指令，因此常量传播需要确定程序的任意执行点处变量的常量性质，即变量的取值。

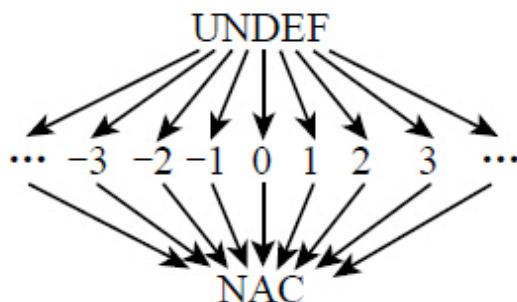


图4-5 整数变量取值半格

图4-5描述了整数变量所有可能取值的半格。其中UNDEF为半格的最大值，表示变量未定义（没有初始化）；NAC为半格的最小值，表示变量确定无常量值；其他值为变量的常量值。半格最大下界运算 \wedge 的性质为（运算符 \wedge 满足交换律）：

- 1) 对于任意值 v ，有 $\text{UNDEF} \wedge v = v$ ，且 $\text{NAC} \wedge v = \text{NAC}$ 。
- 2) 对于常量值 c ，有 $c \wedge c = c$ 。
- 3) 对于不同的常量值 c_1 、 c_2 ，有 $c_1 \wedge c_2 = \text{NAC}$ 。

变量取值半格的最大下界运算用来合并来源于程序不同分支的同一变量的取值，实现数据流分析的交汇运算。

我们根据数据流分析框架四要素讨论常量传播的数据流分析框架。常量传播数据流分析框架定义为：

1) 数据流方向D：常量传播需要沿着代码执行的方向计算变量的值，因此是前向数据流。

2) 值集V：常量传播需要计算函数内所有可见变量的常量值，包括全局变量、参数变量和局部变量，因此常量传播的数据流值是这些变量值的集合。由于是前向数据流，因此边界集合 v_{Entry} 的值是所有可见变量初始值的集合。而初值集合T的所有元素都是UNDEF，即将变量值默认为半格的最大值。边界集合与初值集合都是值集V的元素，因此值集V为可见变量所有可能值集合的全集。

```
1  ConstPropagation::ConstPropagation

2          (DFG*g, SymTab*t, vector<Var*>&paraVar)
3          :dfg(g), tab(t){
4      vector<Var*>glbVars=tab->getGlbVars();
5      int index=0;
6      //变量索引

6      for(unsigned int i=0;i<glbVars.size();++i){                //全局变量

7          Var*var=glbVars[i];
8          var->index=index++;
9          vars.push_back(var);
10         double val=0;
11         if(!var->isBase())val=NAC;
12         else if(!var->unInit())val=var->getVal();
13         boundVals.push_back(val);                                //边界
14     }
15 }
```



```

14     }
15     for(unsigned int i=0;i<paraVar.size();++i){           //参数变量

16         Var*var=paraVar[i];
17         var->index=index++;
18         vars.push_back(var);
19         boundVals.push_back(NAC);                          //边界
值

20     }
21     for(unsigned int i=0;i<dfg->codeList.size();++i){       //局部变量

22         if(dfg->codeList[i]->isDec()){
23             Var*var=dfg->codeList[i]->getArg1();
24             var->index=index++;
25             vars.push_back(var);
26             double val=UNDEF;
27             if(!var->isBase())val=NAC;
28             else if(!var->unInit())val=var->getVal();
29             boundVals.push_back(val);                       //边界
值

30         }
31     }
32     while(index-->0)initVals.push_back(UNDEF);              //初始值

33 }

```

常量传播的构造函数ConstPropagation计算了变量集合vars、边界集合boundVals和初值集合initVals。其中vars中记录的变量对象与boundVals和initVals中的值一一对应。

第4~14行处理全局变量的值。其中第10~12行中，非基本类型变量边界值为NAC，已初始化的基本类型变量边界值为变量初始值，未初始化的全局变量的初值为0，因此边界值为0。

第15~20行处理参数变量的值，函数的参数变量由函数调用者传递的实际参数决定。我们不考虑过程间的代码优化问题，无法确定实际参数的常量性质，因此保守地认为参数变量的边界值为NAC。

第21~31行处理局部变量的值，局部变量都是使用OP_DEC声明的，通过扫描中间代码获得局部变量的定义信息。第26~28行中，对局部变量的处理与全局变量类似，不过未初始化局部变量的边界值为UNDEF。

第32行将初值集合元素全部初始化为UNDEF。

3) 交汇运算 \wedge ：前面讨论了整数取值半格的数值交汇运算，常量传播交汇运算的对象是变量值集合。对于变量值集合A、B，令 $C=A\wedge B$ ，那么对于值集合的任意索引i，总有 $C_i=A_i\wedge B_i$ （其中运算符 \wedge 为整数取值半格的交汇运算）。

```
1  double ConstPropagation::join
   (double left,double right){
2      if(left==NAC||right==NAC)return NAC;                //NAC $\wedge$ 
   v=v
3      else if(left==UNDEF)return right;                    //UNDEF $\wedge$ 
   v=UNDEF
4      else if(right==UNDEF)return left;
5      else if(left==right)return left;                    //c $\wedge$ 
   c=c
6      else return NAC;
   //c1 $\wedge$ 
   c2=NAC
7      }
8  void ConstPropagation::join
   (Block*block){
9      list<Block*>& prevs=block->prevs;                    //前驱
10     vector<double>& in=block->inVals;                     //in
11     for(unsigned int i=0;i<in.size();++i){                //处理
in集合
12         double val=UNDEF;
13         for (list<Block*>::iterator j=prevs.begin();
14             j!=prevs.end();++j){
```

```

//处理前驱

15                      val=join
(val, (*j)->outVals[i]);           //取出前驱
out交并

16                      }
17                      in[i]=val;
18          }
19 }

```

第1~7行的双参数join函数处理变量值的交汇运算，运算规则与变量取值半格的交汇运算特性一致。

第8~19行的单参数join函数处理计算基本块的in集合时的交汇运算。第11行依次计算in集合的每一个元素的值，第13~14行处理基本块block的每一个前驱，第15行取出每一个前驱基本块的out集合对应的索引值，并调用双参数join函数进行交汇运算，第17行将运算结果保存到基本块的in集合。

4) 传递函数集合F: 对于基本块B的传递函数 f_B ，有 $B.out=f_B(B.in)$ 。由于我们更关心每条指令执行前后变量的常量性质，因此将每条指令视为一个基本块，由此原基本块B的传递函数便是指令传递函数 f_s 的复合。对于基本块的指令s，有 $s.out=f_s(s.in)$ 。其中，对于任意可见变量x，其对应的常量性质分别为 $s.out[x]$ 和 $s.in[x]$ 。指令传递函数 f_s 的定义如下。

①如果指令s形式为 $x=c$ ，c为常量，则 $s.out[x]=c$ 。

②如果指令s形式为 $x=y\oplus z$ ， \oplus 为通用运算符。分为以下情况：

a.若 $s.in[y]=c1$ ， $s.in[z]=c2$ ， $c1$ 、 $c2$ 为常量，则 $s.out[x]=c1\oplus c2$ ；

b.若 $s.in[y]=NAC$ 或 $s.in[z]=NAC$ ，则 $s.out[x]=NAC$ ；

c.其他情况， $s.out[x]=UNDEF$ 。

③如果指令s形式为参数传递，且参数为指针类型，则对于任意可见变量 v ， $s.out[v]=NAC$ 。我们保守地认为函数会通过指针参数修改所有可见变量。

④如果指令s形式为 $*x=y$ ，则对于任意可见变量 v ， $s.out[v]=NAC$ 。我们保守地认为对指针内容的修改会影响所有可见变量。

⑤如果指令s形式为 $x=*y$ ，则 $s.out[x]=NAC$ 。

⑥如果指令s形式为`call fun`，则对于任意全局变量 g ， $s.out[g]=NAC$ 。我们保守地认为函数调用修改任意全局变量。

⑦如果指令s形式为 $x=call\ fun$ ，则对于任意全局变量 g ， $s.out[g]=NAC$ 且 $s.out[x]=NAC$ 。

⑧除了以上情况，对于任意可见变量 v ， $s.out[v]=s.in[v]$ 。

指令传递函数 f_s 的实现如下：

```

1 void ConstPropagation::translate
(InterInst*inst,
2     vector<double>& in,vector<double>& out){
3     out=in;
    //默认

4     Operator op=inst->getOp();                                //运
    算符

5     Var*result=inst->getResult();                            //结果

6     Var*arg1=inst->getArg1();                                //参数

1    7     Var*arg2=inst->getArg2();                            //参数

2
8     if(inst->isExpr()){
    //表达式

x=y+z
9         double tmp;
    //保存

out[x]
10         if(op==OP_AS||op==OP_NEG||op==OP_NOT){                //一元运算

11             if(arg1->isLiteral())
//in[y]
12                 tmp=arg1->getVal();
13             else
14                 tmp=in[arg1->index];
15             if(tmp!=UNDEF&&tmp!=NAC){
16                 if(op==OP_NEG)tmp=-tmp;
17                 else if(op==OP_NOT)tmp=!tmp;
18             }
19         }
20         else if(op>=OP_ADD&&op<=OP_OR){                        //二元运
    算

21             double lp,rp;
22             if(arg1->isLiteral())
//in[y]
23                 lp=arg1->getVal();
24             else
25                 lp=in[arg1->index];
26             if(arg2->isLiteral())
//in[z]
27                 if()rp=arg2->getVal();
28             else
29                 rp=in[arg2->index];
30             if(lp==NAC||rp==NAC)tmp=NAC;
//NAC
31             else if(lp==UNDEF||rp==UNDEF)tmp=UNDEF;            //UNDEF
32             else{
//c1+c2
33                 int left=lp,right=rp;
34                 if(op==OP_ADD)tmp=left+right;
35                 else if(op==OP_SUB)tmp=left-right;
36                 else if(op==OP_MUL)tmp=left*right;
37                 else if(op==OP_DIV)

```

```

38             {if(!right)tmp=NAC;else tmp=left/right;}
39             else if(op==OP_MOD)
40             {if(!right)tmp=NAC;else tmp=left%right;}
41             else if(op==OP_GT)tmp=left>right;
42             else if(op==OP_GE)tmp=left>=right;
43             else if(op==OP_LT)tmp=left<right;
44             else if(op==OP_LE)tmp=left<=right;
45             else if(op==OP_EQU)tmp=left==right;
46             else if(op==OP_NE)tmp=left!=right;
47             else if(op==OP_AND)tmp=left&&right;
48             else if(op==OP_OR)tmp=left||right;
49         }
50     }
51     else if(op==OP_GET)
52     //x=*y
53         tmp=NAC;
54         out[result->index]=tmp;
55     //out[x]
56 }
57 else if(op==OP_SET||
58 //x=y
59 op==OP_ARG && !arg1->isBase()){
60 ptr
61     for(unsigned int i=0;i<out.size();++i)
62         out[i]=NAC;
63 //out[v]=NAC
64 }
65 else if(op==OP_PROC){
66 //call f
67     for(unsigned int i=0;i<glbVars.size();++i)
68         out[glbVars[i]->index]=NAC;
69 //out[g]=NAC
70 }
71 else if(op==OP_CALL){
72 //x=call f()
73     for(unsigned int i=0;i<glbVars.size();++i)
74         out[glbVars[i]->index]=NAC;
75 //out[g]=NAC
76     out[result->index]=NAC;
77 //out[x]=NAC
78 }
79 inst->inVals=in;
80 inst->outVals=out;
81 }

```

第3行默认将in集合传递给out集合，后面根据判断特殊情况指令再修改out。

第4~7行获取四元式的基本要素。第8~54行处理形如 $x=c$ 和 $x=y \oplus z$ 的表达式。第55~68行处理其他特殊的指令。69~71行将计算的in和out集合保存到指令对象。

第10~19行处理一元运算表达式赋值、取负、取反。第11~14行取出操作数的常量值，如果arg1是常量则调用getVal取出常量值记录到tmp，否则从in集合取出常量值。第15~18行根据操作符计算表达式结果。

第20~50行处理二元运算表达式。第22~29行取出两个操作数的常量值，分别保存到lp和rp。第30行表示有操作数常量值为NAC，因此计算结果tmp为NAC。第31行表示有操作数常量值为UNDEF，因此计算结果为UNDEF。第32行表示操作数都是常量，需要根据操作符计算表达式结果。

第51~52行处理形如x=*y的表达式，因为无法确定*y的值，所以x的常量值为NAC。

第53行将计算的常量值保存到out集合。

第55~59行处理参数表达式和形如*x=y的表达式，其中第55~56行表明若表达式是指针运算赋值形式或参数为非基本类型，则将out集合所有元素置为NAC。

第60~68处理函数调用，OP_PROC和OP_CALL形式的函数调用都会将全局变量对应的out集合元素置为NAC，而且OP_CALL函数调用会将函数返回值对应的out集合元素置为NAC。

基本块传递函数 f_B 的实现为:

```
1  bool ConstPropagation::translate
   (Block*block){
2      vector<double>in=block->inVals;           //in
3      vector<double>out=in;
   //out=in
4      for(list<InterInst*>::iterator i=block->insts.begin();
5          i!=block->insts.end();++i){           //处
   理指令

6          InterInst*inst=*i;
7          translate
   (inst,in,out);                             //指令传递函数

8          in=out;
   //下条指令的

   in
9      }
10 bool flag=false;
   //out集合是否变化

11 for(unsigned int i=0;i<out.size();++i){
12     if(block->outVals[i]!=out[i]){
13         flag=true;
14         break;
15     }
16 }
17 block->outVals=out;
   //设定

   out
18 return flag;
19 }
```

第2~3行使用基本块的in集合初始化临时变量in和out，作为指令传递函数。

第4行遍历基本块的指令。第7行调用指令传递函数更新out，并将in和out保存到指令对象。第8行将当前指令的out集合作为下一条指令的in集合继续计算。

第10~16行在基本块数据流信息传递结束后，判断基本块的out集合是否发生了变化。

第17行将新计算的out集合更新到基本块的out集合outVals中。

第18行返回传递函数执行后基本块的出口集合是否发生变化。

根据以上讨论的常量传播的数据流分析框架的实现，实现常量传播的数据流分析如下：

```
1 void ConstPropagation::analyse
2 {
3     dfg->blocks[0]->outVals=boundVals;
4     //Entry.out
5     for(unsigned int i=1;i<dfg->blocks.size();++i)
6         dfg->blocks[i]->outVals=initVals;
7     //B.out
8     bool outChange=true;
9     while(outChange){
10        //B.out变化
11
12        outChange=false;
13        for(unsigned int i=1;i<dfg->blocks.size();++i){
14            join(dfg->blocks[i]);
15            //
16
17            if(translate(dfg->blocks[i]))
18                //传递函数
19
20            outChange=true;
21        }
22    }
23 }
```

第2~4行分别使用边界集合boundVals和初值集合initVals初始化基本块的out集合outVals。

第6~13行执行迭代不动点计算，其中第8~12行执行数据流方向的交汇运算join和传递函数translate。当所有基本块的传递函数返回值都是false时，即所有基本块的out信息都不再变化，此时停止迭代计算。

使用analyse进行常量传播数据流分析结束后，所有的指令对象内都保存了执行指令前后所有可见变量的常量性质，即inVals和outVals。根据这两个集合的信息，可以对中间代码指令进行简化。

如图4-6所示，中间代码经过常量传播数据流分析后，产生了对应的数据流信息。图中给出了常量传播优化后的中间代码，常量传播的代码优化规则如下：

1) 常量合并。对于指令s，其四元式为 $\text{result} = \text{arg1} \oplus \text{arg2}$ ，如果 $s.\text{outVals}[\text{result}]$ 是常量c，则使用指令 $\text{result} = c$ 替换原指令。图中指令“a=1”“b=a+2”“b=b+1”按此规则被替换为“a=1”“b=3”“b=4”。

2) 代数化简。对于指令s，其四元式为 $\text{result} = \text{arg1} \oplus \text{arg2}$ ，如果arg1或arg2有一个是常量，且满足运算符 \oplus 的代数化简规则，则将原指令替换为更精简的指令。比如表达式“a=b+0”，可以直接被替换为“a=b”。图4-6中指令“b=a*c”常量传播优化后形式为“b=1*c”，因此可以化简为“b=c”。我们实现的代数化简规则如表4-1所示。

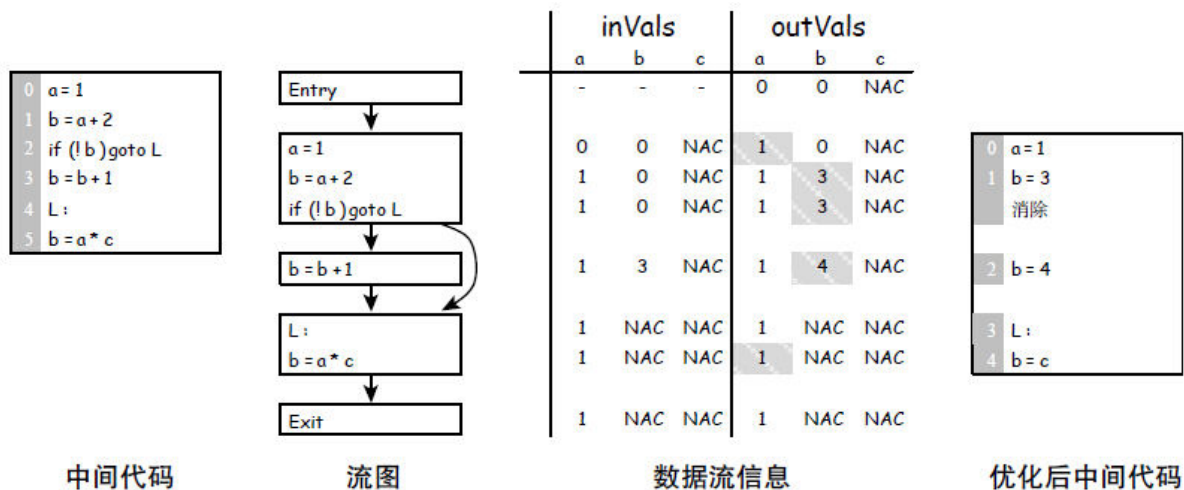


图4-6 常量传播

表4-1 代数化简规则

原表达式	代数化简	原表达式	代数化简
$a=0+b$	$a=b$	$a=0\%b$	$a=0$
$a=b+0$	$a=b$	$a=b\%1$	$a=0$
$a=0-b$	$a=-b$	$a=0\&\&b$	$a=0$
$a=b-0$	$a=b$	$a=b\&\&0$	$a=0$
$a=0*b$	$a=0$	$a=1\&\&b$	$a=(b!=0)$
$a=b*0$	$a=0$	$a=b\&\&1$	$a=(b!=0)$
$a=1*b$	$a=b$	$a=0 b$	$a=(b!=0)$
$a=b*1$	$a=b$	$a=b 0$	$a=(b!=0)$
$a=0/b$	$a=0$	$a=1 b$	$a=1$
$a=b/1$	$a=b$	$a=b 1$	$a=1$

3) 不可达代码消除。对于条件跳转指令s，其形式为if (cond) goto L，如果cond是常量，则需要根据cond条件消除不可能执行的代码分支。如果cond不等于0，则将指令替换为无条件跳转指令jmp L，并解除跳转指令所在基本块与后继基本块的关联。如果cond等于0，则删除条件跳转指令，并解除跳转指令所在基本块与目标基本块的关联。图4-

6中指令“if (! b) goto L”常量传播后形式为“if (! 4) goto L”，该指令不可能执行，因此需要删除，并解除到跳转目标基本块的关联。

按照上述优化规则，常量合并与代数化简实现如下：

```
1 void ConstPropagation::algebraSimplify
2
3 (){
4     for (unsigned int j=0;j<dfg->blocks.size();++j){
5         list<InterInst*>::iterator i;
6         for(i=dfg->blocks[j]->insts.begin();
7             i!=dfg->blocks[j]->insts.end();++i){
8             InterInst*inst=*i;
9             Operator op=inst->getOp();
10            if(inst->isExpr()){
11                double rs;
12                Var*result=inst->getResult();
13                Var*arg1=inst->getArg1();
14                Var*arg2=inst->getArg2();
15                rs=inst->outVals[result->index];
16                if(rs!=UNDEF&&rs!=NAC){ //常量合并
17
18                    Var*newVar=new Var((int)rs);
19                    tab->addVar(newVar);
20                    inst->replace(OP_AS,result,newVar);
21                }
22                else if(op>=OP_ADD&&op<=OP_OR&& //代数化简
23
24                    !(op==OP_AS||op==OP_NEG||op==OP_NOT)){
25                    double lp,rp;
26                    if(arg1->isLiteral())
27                        lp=arg1->getVal();
28                    else
29                        lp=inst->inVals[arg1->index];
30                    if(arg2->isLiteral())
31                        rp=arg2->getVal();
32                    else
33                        rp=inst->inVals[arg2->index];
34                    int left,right;
35                    bool dol=false,dor=false;
36                    if(lp!=UNDEF&&lp!=NAC)
37                        {left=lp;dol=true;}
38                    else if(rp!=UNDEF&&rp!=NAC)
39                        {right=rp;dor=true;}
40                    else continue;
41                    Var* newArg1=NULL;
42                    Var* newArg2=NULL;
43                    Operator newOp=OP_AS;
44                    if(op==OP_ADD){
45                        //z=0+y z=x+0
46                        if(dol&&left==0)newArg1=arg2;
47                        if(dor&&right==0)newArg1=arg1;
48                    }
49                    else if(op==OP_SUB){
50                        //z=0-y z=x-0
51                        if(dol&&left==0)
```

```

46                                     {newOp=OP_NEG;newArg1=arg2;}
47                                     if(dor&&right==0)newArg1=arg1;
48                                 }
49                                 else if(op==OP_MUL){
50
51                                     if(dol&&left==0||dor&&right==0)
52                                         newArg1=SymTab::zero;
53                                     if(dol&&left==1)newArg1=arg2;
54                                     if(dor&&right==1)newArg1=arg1;
55                                 }
56                                 else if(op==OP_DIV){
57                                     //z=0/y z=x/1
58                                     if(dol&&left==0)newArg1=SymTab::zero;
59                                     if(dor&&right==1)newArg1=arg1;
60                                 }
61                                 else if(op==OP_MOD){
62                                     if(dol&&left==0||dor&&right==1)
63                                         newArg1=SymTab::zero;
64                                 }
65                                 else if(op==OP_AND){
66                                     if(dol&&left==0||dor&&right==0)
67                                         newArg1=SymTab::zero;
68                                     if(dol&&left!=0){
69                                         newOp=OP_NE;
70                                         newArg1=arg2;
71                                         newArg2=SymTab::zero;
72                                     }
73                                     if(dor&&right!=0){
74                                         newOp=OP_NE;
75                                         newArg1=arg1;
76                                         newArg2=SymTab::zero;
77                                     }
78                                 }
79                                 else if(op==OP_OR){
80                                     if(dol&&left!=0||dor&&right!=0)
81                                         newArg1=SymTab::one;
82                                     if(dol&&left==0){
83                                         newOp=OP_NE;
84                                         newArg1=arg2;
85                                         newArg2=SymTab::zero;
86                                     }
87                                     if(dor&&right==0){
88                                         newOp=OP_NE;
89                                         newArg1=arg1;
90                                         newArg2=SymTab::zero;
91                                     }
92                                 }
93                                 }
94                                 if(newArg1)
95                                     inst->replace(newOp,result,
96                                     newArg1,newArg2);
97                                 else{
98                                     //代数化简成功
99
100                                     //没法化简，正常传播

```

```

98         if(dol){
99             newArg1=new Var(left);
100             tab->addVar(newArg1);
101             newArg2=arg2;
102         }
103         else if(dor){
104             newArg2=new Var(right);
105             tab->addVar(newArg2);
106             newArg1=arg1;
107         }
108         inst->replace(op,result,
109                     newArg1,newArg2);
110     }
111 }
112 }
113 else if(op==OP_ARG||op==OP_RETV){
114     Var*arg1=inst->getArg1();
115     if(!arg1->isLiteral()){
116         double rs=inst->outVals[arg1->index];
117         if(rs!=UNDEF&&rs!=NAC){
118             Var*newVar=new Var((int)rs);
119             tab->addVar(newVar);
120             inst->setArg1(newVar);
121         }
122     }
123 }
124 }
125 }
126 }

```

第2~7行取出流图基本块的每一条指令进行处理。第8~112行处理表达式的常量传播。第113~123行处理参数指令和返回指令的常量传播。

第9~12行取出四元式的基本元素。第13~18行处理常量合并，将result的常量值取出存入rs变量，并将原指令替换为指令result=rs。

第19~111行处理代数化简。第21~29行将arg1和arg2的常量值取出到lp和rp。第30~36行判定哪个操作数是常量，将信息记录到dol和dor中，将操作数的常量值保存到left和right中。

第37~39行使用newArg1和newArg2记录新的操作数，使用newOp记录新的操作符，其初值为赋值运算符OP_AS。

第40~43行处理加法操作的代数化简，若有操作数为0，便将另一个操作数作为newArg1。其他算术运算的代数化简基本类似，在此不再赘述。需要注意的是减法操作代数化简时，可能更改newOp为OP_NEG取负指令。

第64~78行处理逻辑与操作数的代数化简。若有操作数为0，便将newArg1记录为SymTab: : zero，表示表达式结果为0。若有操作数为1，则需要将另一个操作数转化为布尔值，即将newOp更改为OP_NE，另一个操作数保存到newArg1，newArg2记录为SymTab: : zero。例如表达式“a=1||b”，经过代数化简后转化为“a= (b! =0)”，而不是“a=b”。逻辑或运算的代数化简与此类似。

常量传播的不可达代码消除算法如下:

```

1 void ConstPropagation::condJumpOpt
2 {
3     for (unsigned int j = 0; j < dfg->blocks.size(); ++j){
4         list<InterInst*>::iterator i,k;
5         for(i=dfg->blocks[j]->insts.begin(),k=i;
6             i!=dfg->blocks[j]->insts.end();i=k){
7             ++k;
8             InterInst*inst=*i;
9             if(inst->isJcond()){
10                 Operator op=inst->getOp();
11                 InterInst*tar=inst->getTarget();
12                 Var*arg1=inst->getArg1();
13                 double cond;
14                 if(arg1->isLiteral())cond=arg1->getVal();
15                 else cond=inst->inVals[arg1->index];
16                 if(cond==NAC||cond==UNDEF)continue;
17                 if(op==OP_JT&&cond==0||op==OP_JF&&cond!=0){
18                     inst->block->insts.remove(inst);
19                 }
20             }
21         }
22     }
23 }

```

```

18             if(dfg->blocks[j+1]!=tar->block)
19                 dfg->delLink(inst->block, tar->block);
20         }
21     else if(op==OP_JT&&cond!=0 || op==OP_JF&&cond==0){
22         inst->replace(OP_JMP, tar);
23         if(dfg->blocks[j+1]!=tar->block)
24             dfg->delLink(inst->block, dfg->blocks[j+1]);
25     }
26 }
27 }
28 }
29 }

```

第2~6行遍历流图基本块的所有指令，迭代器*k*始终指向迭代器*i*的下一个位置，以防止遍历过程删除*i*指向的指令后无法继续迭代的情况。

第9~11行取出四元式的基本元素。第12~15行取出条件变量的常量值，保存到*cond*。

第16~20行处理跳转条件不满足的情况。首先删除跳转指令，然后调用*delLink*解除当前指令所在基本块与跳转目标基本块的关联。

第21~25行处理跳转条件总是满足的情况。首先将跳转指令替换为无条件跳转指令*OP_JMP*，然后调用*delLink*解除当前指令所在基本块与后继基本块的关联。

函数*delLink*用于解除两个基本块之间的关联，需要对流图进行操作。

```

1 void DFG::delLink
  (Block*begin,Block*end){
2     if(begin){
3         begin->succs.remove(end);
4         end->prevs.remove(begin);

```



```

5         }
6         release(end);                                     //递
    归解除关联

7     }
8     void DFG::release

(Block*block){
9         if(!reachable(block)){                           //块不可达

10             list<Block*> delList;
11             list<Block*>::iterator i;
12             for(i=block->succs.begin();i!=block->succs.end();++i){
13                 delList.push_back(*i);                   //记录所有
    后继

14             }
15             for(i=delList.begin();i!=delList.end();++i){
16                 block->succs.remove(*i);
17                 (*i)->prevs.remove(block);
18             }
19             for(i=delList.begin();i!=delList.end();++i){
20                 release(*i);                             //递
    归处理后继

21         }
22     }
23 }
24 bool DFG::reachable

(Block*block){
25     resetVisit();
26     return __reachable(block);
27 }
28 bool DFG::__reachable

(Block*block){
29     if(block==blocks[0])return true;                     //到达入口

30     else if(block->visited)return false;                 //访问过了

31     block->visited=true;                                  //设定访问标记

32     bool flag=false;
33     list<Block*>::iterator i;
34     for(i=block->prevs.begin();i!=block->prevs.end();++i){
35         Block*prev=*i;                                    //
    每个前驱

36         flag=__reachable(prev);                           //递归测试

37         if(flag)break;
38     }
39     return flag;
40 }

```

第1~7行的delLink函数删除基本块begin到end的关联，即将end从begin的后继中删除，将begin从end的前驱中删除。基本块关联解除后，还需要测试end是否可达。如果end不可达，那么从end出发的关联也是无效的，也需要删除，这是一个递归的过程。

第8~23行的release函数用于处理基本块不可达时删除无效的基本块关联。第9行调用reachable测试基本块block是否从Entry开始可达。如果基本块不可达，第11~14行将block的后继添加到列表delList。第15~18行遍历delList按照第1~7行相似的方式解除block与后继基本块的管理。第19~21行再次遍历delList，递归调用release处理block的后继基本块。

第28~40行的__reachable函数用于测试基本块block是否从Entry块开始可达。第29行表示block就是Entry块，返回true。第30行表示block已经被访问过了，返回false。第34~38行处理block的前驱，向前进行深度搜索。第36行递归调用__reachable处理各个前驱，只要有一个前驱可达，则block可达。

第24~27行的reachable函数对__reachable进行封装，即在调用__reachable函数之前，调用resetVisit函数将所有基本块的visit访问标记置为false，以防止多次调用__reachable函数导致访问标记被污染。

4.2.2 复写传播

如果说常量传播是将常量传递到表达式的操作数中，那么复写传播则是将变量传递到表达式的操作数中。复写传播分析变量值的复制轨迹，以发现等值变量集合，并在表达式中尽可能使用同一个变量代替原本表达式的操作数。例如中间代码

```
b=a;c=b;d=c+1
```

其中“**b=a**”“**c=b**”称为复写表达式，经过复写传播后，上述代码被转化为：

```
b=a;c=a;d=a+1
```

我们发现原中间代码为了计算变量**d**的值，不得不依次计算变量**b**和**c**的值。但是，经过复写传播的处理，使得对变量**b**和**c**的计算变得冗余，使之成为无效代码（死代码）。死代码消除会将该类代码从中间代码中删除，因此复写传播从一定程度上来说是为死代码消除服务的。

对于复写传播，其数据流分析框架定义如下。

1) 数据流方向D: 复写传播需要沿着代码执行的方向分析复写表达式, 因此是前向数据流。

2) 值集V: 复写传播分析的是代码中的复写表达式, 因此复写传播的数据流值是复写表达式集合。由于是前向数据流, 因此边界集合vEntry的值为空集。而初值集合T为所有复写表达式的全集。边界集合与初值集合都是值集V的元素, 因此值集V为复写表达式的幂集。

3) 交汇运算 \wedge : 当基本块具有多个前驱时, 基本块入口处的复写表达式集合为前驱集合出口处复写表达式集合的交集, 因此交汇运算为集合交运算 \cap 。

4) 传递函数: 与常量传播的传递函数类似, 这里仍将指令看作独立的基本块。复写表达式的一般形式为“ $x=y$ ”, 与赋值表达式的形式完全相同。如果表达式运算修改了x或y, 称为指令杀死了复写表达式“ $x=y$ ”。对于赋值表达式“ $x=y$ ”, 称为指令产生了复写表达式“ $x=y$ ”, 同时该指令杀死了包含x的复写表达式。指令产生的复写表达式集合记为s.gen, 指令杀死的复写表达式集合记为s.kill, 因此指令的传递函数fs定义为: $s.out = (s.in - s.kill) \cup s.gen$ (其中‘-’为集合差集运算, \cup 为集合并集运算)。

复写传播初始化阶段, 需要统计所有的复写表达式, 并计算指令的gen和kill集合。

```

1 CopyPropagation::CopyPropagation
(DFG*g):dfg(g){
2     dfg->toCode(optCode);
3     int j=0;
4     for(list<InterInst*>::iterator i=optCode.begin();
5         i!=optCode.end();++i){
6         InterInst*inst=*i;
7         Operator op=inst->getOp();
8         if(op==OP_AS)copyExpr.push_back(inst);
9     }
10    U.init(copyExpr.size(),1);
11    E.init(copyExpr.size(),0);
12    G.init(copyExpr.size(),0);
13    vector<Var*>glbVars=tab->getGlbVars();
14    for(unsigned int i=0;i<glbVars.size();++i){
15        for(unsigned int j=0;j<copyExpr.size();j++){
16            if(glbVars[i]==copyExpr[j]->getResult())
17                ||glbVars[i]==copyExpr[j]->getArg1())
18                G.set(i);
19        }
20    }
21    for(list<InterInst*>::iterator i=optCode.begin();
22        i!=optCode.end();++i){
23        InterInst*inst=*i;
24        inst->copyInfo.gen=E;
25        inst->copyInfo.kill=E;
26        Var*rs=inst->getResult();
27        Operator op=inst->getOp();
28        if(op==OP_SET||op==OP_ARG&&!inst->getArg1()->isBase())
29            inst->copyInfo.kill=U;
30        else if(op==OP_PROC||op==OP_CALL)
31            inst->copyInfo.kill=G;
32        if(op==OP_AS&&op!=OP_OR||op==OP_GET||op==OP_CALL){
33            for(unsigned int i=0;i<copyExpr.size();i++){
34                if(rs==copyExpr[i]->getResult()
35                    ||rs==copyExpr[i]->getArg1())
36                    inst->copyInfo.kill.set(i);
37                if(copyExpr[i]==inst)
38                    inst->copyInfo.gen.set(i);
39            }
40        }
41    }
}

```

第2~9行从流图中提取所有的中间代码到`optCode`，并遍历中间代码，记录所有赋值运算表达式，即复写表达式到`copyExpr`。

第10~19行初始化变量U、E和G，它们分别表示复写表达式的全集、空集和包含全局变量的复写表达式集合。这些集合的大小与`copyExpr`大小相等，集合元素取值为0或1，表示`copyExpr`对应索引处

的复写表达式是否存在。其中第18行的set函数是将索引i的集合元素置为1。

第20~40行计算指令的gen和kill集合，其中第23~24行将gen和kill初始化为空集E。

第27~28行处理指针运算的赋值指令和指针参数指令，它们可能杀死所有的复写表达式，因此将kill置为全集U。

第29~30行处理函数调用指令，它们可能杀死所有包含全局变量的复写表达式，因此将kill置为G。而对有返回值的函数，调用指令OP_CALL的返回值放在后面处理。

第31~39处理所有修改结果变量rs的指令，并与复写表达式集合copyExpr进行比对。

第33~35行表示指令运算结果rs是复写表达式copyExpr[i]的一部分，即指令杀死了复写表达式copyExpr[i]，因此将指令的kill集合索引i的元素置为1。

第36~37行表示该指令为赋值表达式，产生了复写表达式copyExpr[i]，因此将指令的gen集合索引i的元素置为1。赋值表达式杀死的复写表达式已经在第33~35行处理。

产生集合**gen**和杀死集合**kill**初始化完毕后，便可以实现传递函数的功能。

```
1  bool CopyPropagation::translate
   (Block*block){
2      Set tmp=block->copyInfo.in;
3      for(list<InterInst*>::iterator i=block->insts.begin();
4          i!=block->insts.end();++i){
5          InterInst*inst=*i;
6          Set& in=inst->copyInfo.in;
7          Set& out=inst->copyInfo.out;
8          in=tmp;
9          out=(in-inst->copyInfo.kill)
10             |inst->copyInfo.gen;
11             tmp=out;
12     }
13     bool flag=tmp!=block->copyInfo.out;
14     block->copyInfo.out=tmp;
15     return flag;
16 }
```

第2行将**tmp**集合初始化为基本块的入口集合**B.in**。第3~12行按序处理基本块内的指令。

第8行将**tmp**记录到指令的入口集合**s.in**。第9行根据指令的传递函数计算指令的出口集合**s.out**。指令传递函数使用了运算符‘|’和‘-’用于表示集合的并集和差集运算，这些运算符已经被**Set**类重载。

第11行将指令出口集合**s.out**更新到**tmp**，作为下条指令的入口集合。

第13行判断基本块的出口集合**B.out**是否发生更新。第14行更新基本块的出口集合。

根据复写传播的传递函数实现复写传播的数据流分析如下。

```
1 void CopyPropagation::analyse
2 {
3     dfg->blocks[0]->copyInfo.out=E;
4     //Entry.out=E
5     for(unsigned int i=1;i<dfg->blocks.size();++i){
6         dfg->blocks[i]->copyInfo.out=U;           //B.out=U
7     }
8     bool change=true;
9     while(change){
10        //B.out集合发生变化
11
12        change=false;
13        for(unsigned int i=1;i<dfg->blocks.size();++i){
14            if(!dfg->blocks[i]->canReach)continue;
15            Set tmp=U;
16            list<Block*>::iterator j;
17            for(j=dfg->blocks[i]->prevs.begin();
18                j!=dfg->blocks[i]->prevs.end();++j){
19                tmp=tmp & (*j)->copyInfo.out;
20            }
21            dfg->blocks[i]->copyInfo.in=tmp;
22            if(translate(dfg->blocks[i]))
23                change=true;
24        }
25    }
26 }
```

第2~5行初始化常量传播基本块的出口集合，其中Entry块出口集合初始化为空集E，其他基本块出口集合初始化为全集U。

第7~20行进行迭代不动点计算。其中第10行跳过不可达的基本块，第15行使用集合交集运算‘&’合并前驱基本块的出口集合到B.in，第18行调用传递函数计算B.out。

复写传播数据流分析结束后，每条指令的入口集合内保存了有效的复写表达式集合。

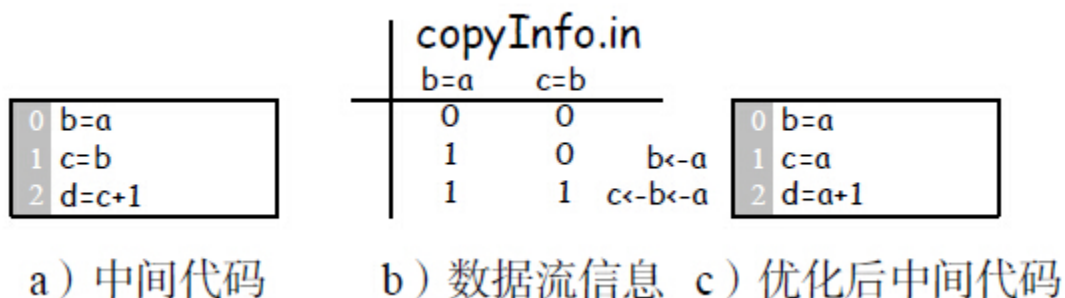


图4-7 复写传播

本节开始的例子经过复写传播数据流分析后的数据流信息如图4-7所示。在处理指令“c=b”时，入口复写表达式集合为{b=a}，复写传播链为“b←a”因此使用变量a替换变量b，得到指令“c=a”。在处理指令“d=c+1”时，入口复写表达式集合为{b=a, c=b}，复写传播链为“c←b←a”，因此使用变量a替换变量c的使用，得到指令“d=a+1”。

根据复写链查找变量的实现为：

```
1 Var* CopyPropagation::find
  (Set& in,Var*var){
2     if(!var)return NULL;
3     for(unsigned int i=0;i<copyExpr.size();i++){
4         if(in.get(i)){
5             Var*rs=copyExpr[i]->getResult();
6             Var*arg1=copyExpr[i]->getArg1();
7             if(var!=rs)continue;
8             if(var!=rs||rs==arg1)break;
9             return find
              (in,arg1);
10        }
11    }
12    return var;
13 }
```

参数in表示指令入口复写表达式集合，var为待查找的变量。

第3~4行遍历入口集合的复写表达式，Set的get函数测试索引为i的元素是否存在。

第5~6行取出复写表达式的结果rs和参数arg1。

第7行判断如果待查找的变量不是复写表达式的结果变量，则继续向后查找。

第8行判断如果复写表达式的形式为“x=x”，则停止查找过程，避免无限查找。

第9行将arg1作为待查找变量继续递归查找。

使用find函数总能找到边var复写传播的“源头”，即复写值的来源。

使用复写传播数据流信息对代码优化的实现为：

```
1 void CopyPropagation::propagate
2 (){
3     analyse();
4     for(list<InterInst*>::iterator i=optCode.begin();
5         i!=optCode.end();++i){
6         InterInst*inst=*i;
7         Var* rs=inst->getResult();
8         Operator op=inst->getOp();
9         Var*arg1=inst->getArg1();
10        Var*arg2=inst->getArg2();
11        InterInst*tar=inst->getTarget();
12        if(op==OP_SET){
13            Var*newRs=find(inst->copyInfo.in,rs);
14            inst->replace(op,newRs,arg1);
15        }
16        else if(op>=OP_AS&&op<=OP_GET&&op!=OP_LEA){
17            Var*newArg1=find(inst->copyInfo.in,arg1);
18            Var*newArg2=find(inst->copyInfo.in,arg2);
19            inst->replace(op,rs,newArg1,newArg2);
```

```
19         }
20     else if (op==OP_JT || op==OP_JF || op==OP_ARG || op==OP_RETV){
21         Var*newArg1=find(inst->copyInfo.in, arg1);
22         inst->setArg1(newArg1);
23     }
24 }
25 }
```

第2行调用**analyse**函数进行复写传播数据流分析。

第3~4行处理每一条中间代码指令。

第11~14行处理指针运算赋值指令“***arg1=result**”，调用**find**查找**result**的复写值来源**newRs**，然后更新指令。

第15~19行处理一般的运算指令，其中排除了取址运算指令**OP_LEA**，这是因为取址运算是针对变量的，而非变量的值。

第16~17行取出操作数的复写值来源，然后更新指令。

第20~22行处理其他运算指令，同样将**arg1**的复写值来源更新到指令。

4.2.3 死代码消除

前面讨论过，复写传播从一定意义上说是为死代码消除算法服务的。所谓死代码，就是对程序计算结果没有任何影响的代码。死代码消除就是发现这样的死代码，并将之从代码中删除。死代码消除是基于变量的活跃性数据流分析过程的，变量活跃性分析是为了发现某条指令执行后，哪些变量还会被使用。如果对变量值的修改（称为定值）指令执行后，该变量不会在以后再被使用，那么便认为这条指令是无效的，即死代码。如图4-7所示复写传播优化后的中间代码，在指令“`b=a`”执行后，变量不会再被使用，因此该指令为死代码，同样的指令“`c=b`”也是死代码。而在复写传播优化之前，所有的变量都会被直接或间接地使用，因此不存在死代码，这说明了复写传播优化对于死代码消除的必要性。

活跃变量数据流分析框架的定义如下。

1) 数据流方向**D**：活跃变量分析计算将来要被使用的变量集合，与代码执行顺序相反，因此是逆向数据流。

2) 值集**V**：活跃变量分析的是所有的可见变量，包括全局变量、参数变量和局部变量，这与常量传播分析的对象相同。由于是逆向数据流，边界集合**vExit**的值为空集，因为**Exit**块不会使用任何变量。初

值集合？也是空集，因为交汇运算是集合并集运算。边界集合和初值集合都是值集 V 的元素，因此值集 V 为可见变量的幂集。

3) 交汇运算 \wedge ：当基本块具有多个后继时，基本块出口处的活跃变量集合为后继集合入口处活跃变量集合的并集，因此交汇运算为集合并运算 \cup 。

4) 传递函数：与前面描述的数据流问题的传递函数类似，这里仍将指令看作独立的基本块。在复写传播数据流中，定义了指令的产生复写表达式集合 $s.gen$ 和杀死复写表达式集合 $s.kill$ 。类似地，活跃变量分析数据流也为指令定义了两个集合：指令定值变量集合 $s.def$ 和指令使用变量集合 $s.use$ 。对于通用的指令形式 $x=f(y)$ ，其中 x 为指令计算结果， f 为指令计算操作， y 为指令参数，我们称为该指令是对 y 的使用，对 x 的定值。那么 x 属于指令的定值变量集合 $s.def$ ， y 属于指令的使用变量集合 $s.use$ 。如果 y 和 x 是同一个变量，即 $x=f(x)$ ，这里规定 x 属于指令的使用集合 $s.use$ 。指令的传递函数 fs 定义为： $s.in = (s.out - s.def) \cup s.use$ 。

活跃变量分析初始化阶段，需要统计所有的可见变量，并计算指令的 def 和 use 集合。

```
1 LiveVar::LiveVar
  (DFG*g, SymTab*t, vector<Var*>&paraVar)
2     :dfg(g), tab(t){
3     varList=tab->getGlbVars();
4     int glbNum=varList.size();
```

```

5      for(unsigned int i=0;i<paraVar.size();++i)
6          varList.push_back(paraVar[i]);
7      dfg->toCode(optCode);
8      for(list<InterInst*>::iterator i=optCode.begin();
9          i!=optCode.end();++i){
10         InterInst*inst=*i;
11         Operator op=inst->getOp();
12         if(op==OP_DEC)varList.push_back(inst->getArg1());
13     }
14     U.init(varList.size(),1);
15     E.init(varList.size(),0);
16     G=E;
17     for(int i=0;i<glbNum;i++)G.set(i);
18     for(unsigned int i=0;i<varList.size();i++)
19         varList[i]->index=i;
20     for(list<InterInst*>::iterator i=optCode.begin();
21         i!=optCode.end();++i){
22         InterInst*inst=*i;
23         inst->liveInfo.use=E;
24         inst->liveInfo.def=E;
25         Var*rs=inst->getResult();
26         Operator op=inst->getOp();
27         Var*arg1=inst->getArg1();
28         Var*arg2=inst->getArg2();
29         if(op>=OP_AS&&op<=OP_LEA){
30             inst->liveInfo.use.set(arg1->index);
31             if(arg2)
32                 inst->liveInfo.use.set(arg2->index);
33             if(rs!=arg1 && rs!=arg2)
34                 inst->liveInfo.def.set(rs->index);
35         }
36         else if(op==OP_SET)
37             inst->liveInfo.use.set(rs->index);
38         else if(op==OP_GET)
39             inst->liveInfo.use=U;
40         else if(op==OP_RETV)
41             inst->liveInfo.use.set(arg1->index);
42         else if(op==OP_ARG){
43             if(arg1->isBase())
44                 inst->liveInfo.use.set(arg1->index);
45             else
46                 inst->liveInfo.use=U;
47         }
48         else if(op==OP_CALL||op==OP_PROC){
49             inst->liveInfo.use=G;
50             if(rs&&rs->getPath().size(>1)
51                 inst->liveInfo.def.set(rs->index);
52         }
53         else if(op==OP_JF||op==OP_JT)
54             inst->liveInfo.use.set(arg1->index);
55     }
56 }

```

第3~13行将全局变量、参数变量和局部变量保存到变量集合 `varList`。第14~19行初始化了数据流值的全集、空集、全局变量集合和变量在变量集合 `varList` 的索引。

第20~55行计算指令的def和use集合。第29~35行处理一般的运算指令，即将arg1和arg2添加到指令的use集合，如果rs不等于arg1和arg2，则将rs添加到指令的def集合。

第36~37行处理指针运算赋值指令“*arg1=result”，我们可以确定result一定会被使用。arg1指向的变量无法确定，但不能认为产生了对任意变量的定值，否则会消除除了result的所有变量的活跃信息，导致正常的代码被处理为死代码。因此，在无法确定指令的定值变量集合时，保守地认为没有变量被定值。

第38~39行处理指令运算取值指令“result=*arg1”，这里无法确定result是否被定值，因为arg1有可能指向result。arg1指向的变量无法确定，在不能确定指令的使用变量集合时，保守地认为所有的变量被使用。

第40~41行处理函数返回指令，将arg1添加到指令的使用变量集合。

第42~47行处理参数指令，如果参数是基本类型，将之添加到使用变量集合，如果参数是非基本类型，则认为所有的变量会被使用。

第48~52行处理函数调用指令，保守认为函数调用会使用所有的全局变量。如果函数有返回值，且返回值不是全局变量，则将返回值添加到定值变量集合。

第53~54行处理跳转指令，将arg1添加到使用变量集合。

根据指令的def和use集合，实现基本块的传递函数如下。

```
1  bool LiveVar::translate
   (Block*block){
2      Set tmp=block->liveInfo.out;
3      for(list<InterInst*>::reverse_iterator
4          i=block->insts.rbegin();
5          i!=block->insts.rend();++i){
6          InterInst*inst=*i;
7          if(inst->isDead)continue;
8          Set& in=inst->liveInfo.in;
9          Set& out=inst->liveInfo.out;
10         out=tmp;
11         in=inst->liveInfo.use | (out-inst->liveInfo.def);
12         tmp=in;
13     }
14     bool flag=tmp!=block->liveInfo.in;
15     block->liveInfo.in=tmp;
16     return flag;
17 }
```

活跃变量分析的传递函数和复写传播的传递函数基本类似，只不过对基本块内的指令是使用reverse_iterator逆序遍历的，这是因为活跃变量数据流是逆向的。

另外，第7行跳过了死代码指令，后面的死代码消除算法会对此作出解释。

根据活跃变量分析的传递函数实现活跃变量数据流分析如下。

```
1  void LiveVar::analyse
   (){
2      dfg->blocks[dfg->blocks.size()-1]->liveInfo.in=E;
   //Exit.in
3      for(unsigned int i=0;i<dfg->blocks.size()-1;++i){
4          dfg->blocks[i]->liveInfo.in=E;
   //B.in=E
5      }
```

```

6      bool change=true;
7      while(change){
8          change=false;
9          for(int i=dfg->blocks.size()-2;i>=0;--i){           //
逆序

10              if(!dfg->blocks[i]-
canReach)continue;
11              Set tmp=E;
12              list<Block*>::iterator j;
13              for(j=dfg->blocks[i]->succs.begin();
14                  j!=dfg->blocks[i]->succs.end();++j){
15                  tmp=tmp | (*j)->liveInfo.in;
16              }
17              dfg->blocks[i]->liveInfo.out=tmp;
18              if(translate(dfq->blocks[i]))
19                  change=true;
20          }
21      }
22 }

```

第2~5行将Exit块的入口集合初始化为空集，将其他基本块的入口集合也初始化为空集。

第7~21行进行迭代不动点计算，第9行逆序处理基本块，第10行跳过不可达基本块。

第15行使用集合并运算“|”以将前驱基本块的出口集合合并到B.out，第17行调用传递函数来计算B.in。

活跃变量分析结束后，指令的出口集合保存了所有将要使用的变量集合。如果对变量的定值指令的出口集合中不包含该变量，则将该指令标记为死代码。

```

1  void LiveVar::eliminateDeadCode
(int stop=false){
2      if(stop){
3          for(list<InterInst*>::iterator i=optCode.begin();
4              i!=optCode.end();++i){

```

```

5             InterInst*inst=*i;
6             Operator op=inst->getOp();
7             if(inst->isDead||op==OP_DEC)continue;
8             Var*rs=inst->getResult();
9             Var*arg1=inst->getArg1();
10            Var*arg2=inst->getArg2();
11            if(rs)rs->live=true;
12            if(arg1)arg1->live=true;
13            if(arg2)arg2->live=true;
14        }
15        for(list<InterInst*>::iterator i=optCode.begin();
16            i!=optCode.end();++i){
17            InterInst*inst=*i;
18            Operator op=inst->getOp();
19            if(op==OP_DEC){
20                Var*arg1=inst->getArg1();
21                if(!arg1->live)inst->isDead=true;
22            }
23        }
24        return ;
25    }
26    stop=true;
27    analyse();
28    for(list<InterInst*>::iterator i=optCode.begin();
29        i!=optCode.end();++i){
30        InterInst*inst=*i;
31        if(inst->isDead)continue;
32        Var*rs=inst->getResult();
33        Operator op=inst->getOp();
34        Var*arg1=inst->getArg1();
35        Var*arg2=inst->getArg2();
36        if(op>=OP_AS&&op<=OP_LEA||op==OP_GET){
37        if(rs->getPath().size()==1)continue;
38        if(!inst->liveInfo.out.get(rs->index)
39            ||op==OP_AS&&rs==arg1){
40            inst->isDead=true;
41            stop=false;
42        }
43        }
44        else if(op==OP_CALL){
45            if(!inst->liveInfo.out.get(rs->index))
46                inst->callToProc();
47        }
48    }
49    eliminateDeadCode(stop);
50 }

```

第2~25行处理变量声明指令OP_DEC，第26~49行对死代码进行标记。

第26行设定stop标记，期望算法可以终止。第27行调用analyse进行活跃变量分析。

第28~48行遍历所有指令，标记死代码。第49行递归调用 `eliminateDeadCode` 进行死代码消除。

第36~43行处理一般的变量定值指令，第37行跳过对全局变量的定值指令，第38行表示被赋值的变量 `rs` 不在指令的出口集合中，第39行识别形如“`x=x`”的指令，第40行将指令标记为死代码。第41行设定 `stop` 为 `false`，需要再次运行死代码消除算法，这是因为消除死代码后，可能使原本的活跃变量数据流信息发生变化。

第44~46行处理函数调用指令，如果函数返回值没有被使用的话，则将有返回值函数调用指令 `OP_CALL` 修正为无返回值函数调用指令 `OP_PROC`。

当没有新的死代码被标记时，`stop` 被设置为 `true`，此时处理变量的声明指令。

第3~14行将所有的非死代码和非 `OP_DEC` 指令的结果和参数变量标记为活跃的，第15~23行处理 `OP_DEC` 指令，如果指令的操作数 `arg1` 变量不是活跃的，则将 `OP_DEC` 指令标记为死代码，达到清洗变量声明指令的目的。

4.3 寄存器分配

CPU内的寄存器有比内存更高的访问效率，但是寄存器资源非常有限，如何合理地利用寄存器资源，提高代码的性能，是编译优化关心的问题。

4.3.1 图着色算法

当代码中使用的变量个数小于寄存器个数时，我们可以为所有的变量各指定一个寄存器，实现变量的高效访问。一般情况下，寄存器的个数远小于变量个数，当寄存器保存了一个变量后，就不能再保存其他变量。但是也有例外，如果保存在寄存器的变量在以后不会再被用到（不再活跃），那么后来的变量便可以保存在这个寄存器中。可以看出，变量的活跃性信息对寄存器的分配至关重要。

如果两个变量永远不能保存在同一个寄存器，则称这两个变量是相互冲突的。在变量的活跃性分析结束后，每条指令的出口集合内保存的变量都是活跃的，即在将来会被用到。同时活跃的变量是不能保存在同一寄存器的，因此是相互冲突的。

图4-8描述了左侧中间代码经活跃变量分析后，产生的活跃变量信息，即指令的out集合。Entry指令出口集合为{a, b, d, f}，表示这四个变量是同时活跃的，它们是相互冲突的。如果以变量为节点，以变量之前的冲突关系作为边，构造图数据结构（冲突图），便可以表达所有变量之间的冲突关系。

图4-9描述了根据图4-8的变量活跃性信息构造的冲突图。我们发现，对于任意指令的出口集合的变量，它们的冲突关系在冲突图内表

现为一个完全子图。

中间代码	活跃变量out集合
Entry	{a, b, d, f}
0 c=a+b	{c, d, f}
1 e=c-d	{c, e, f}
2 e=e*c	{e, f}
3 f=f/e	{f}
4 return f goto L	{}
Exit	-

图4-8 变量活跃性信息

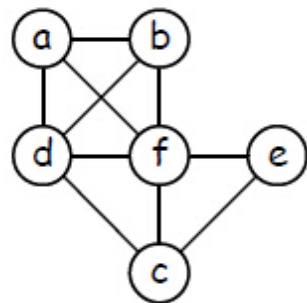


图4-9 冲突图

经典的寄存器分配算法使用的是图着色算法，图着色算法的目的是为图节点分配颜色，并保证相关联的节点的颜色不能相同。图着色算法的计算对象是冲突图，对冲突图进行着色后，每个节点都获得了颜色，且相关节点的颜色互不相同。如果将颜色看作寄存器，由于冲突图的节点表示代码中的变量，因此对冲突图的图着色实际是完成了变量的寄存器分配。

由于图着色问题是NP问题，即不存在多项式时间的算法找到图的最佳着色方案。但是，存在多项式时间的近似最佳图着色方案，基本思想如下：

- 1) 选取度最大的节点进行着色。
- 2) 与被着色节点相连的节点不能再着相同的颜色。
- 3) 将被着色节点及其关联边从图中删除。
- 4) 重复以上过程，直到节点全部着色或没有颜色可以使用为止。

如图4-10所示图着色算法的例子，假设我们有充足的颜色可以使用，颜色编号从1开始。

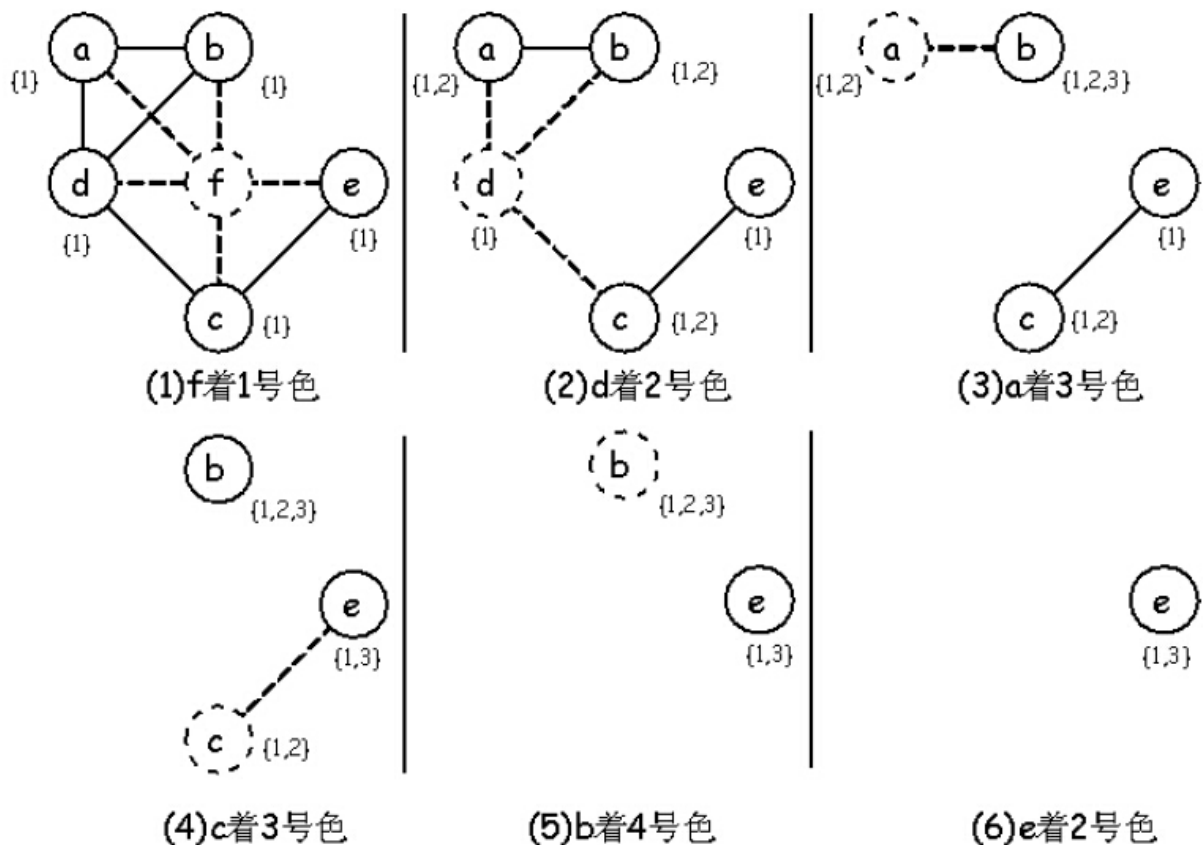


图4-10 图着色

对于冲突图，选取最大度节点f，着颜色1，节点a、b、c、d、e记录不能再着1号色，然后删除节点f及其关联边。接着选择最大度节点d，d不能着1号色，选取颜色2对d着色，节点a、b、c不能着2号色，然后删除节点d及其关联边。以此类推，将节点a着3号色，节点c着3号色，节点b着4号色，节点e着2号色。

我们发现原本图4-10中有6个节点，而使用4种颜色便可以完成图的着色。换句话说，代码中的6个变量可以使用4个寄存器进行保存。如

果可用的寄存器只有三个，那么按照上述图着色算法，无法为变量d分配寄存器，此时d只能保存在内存中。

冲突图相关数据结构定义如下。

```
1 struct Node
2 {
3     Var*var; //节点对应的变量
4     int degree; //节点的度
5     int color; //节点颜色
6     Set exColors; //节点排斥的颜色
7     vector<Node*> links; //关联边
8 };
9 class CoGraph //冲突图
10 {
11     struct node_less{ //节点度比较
12         bool operator()(Node*left,Node*right){
13             return left->degree<=right->degree;
14         };
15     };
16     vector<Node*> nodes; //节点序列
17 };
18
```

类型Node表示冲突图的节点，其字段var记录节点对应的变量，字段degree实时记录节点的度。字段color记录寄存器分配后节点的颜色，即寄存器的编号，初始化为-1表示没有分配寄存器。字段exColor表示节点在寄存器分配过程中不能使用的颜色集合。字段links表示节点的

相邻节点集合。类型CoGraph表示冲突图，node_less用于比较两个节点度的大小，字段nodes记录冲突图的所有节点。

根据变量活跃性信息构造冲突图的实现如下。

```
1  CoGraph::CoGraph
   (list<InterInst*>&optCode,
2      vector<Var*>&para, LiveVar*lv, Fun*f){
3      fun=f;
4      this->optCode=optCode;
5      this->lv=lv;
   //活跃变量信息

6      U.init(regNum,1);
   //颜色集合全集

7      E.init(regNum,0);
   //颜色集合空集

8      for(unsigned int i=0;i<para.size();++i)                //参数

9          varList.push_back(para[i]);
10     for(list<InterInst*>::iterator i=optCode.begin();
11         i!=optCode.end();++i){
12         InterInst*inst=*i;
13         Operator op=inst->getOp();
14         if(op==OP_DEC){
   //局部变量

15             Var* arg1=inst->getArg1();
16             varList.push_back(arg1);
17         }
18         if(op==OP_LEA){
   //取址

19             Var* arg1=inst->getArg1();
20             if(arg1)arg1->inMem=true;
21         }
22     }
23     Set& liveE=lv->getE();
24     Set mask=liveE;
25     for(unsigned int i=0;i<varList.size();i++)
26         mask.set(varList[i]->index);
27     for(unsigned int i=0;i<varList.size();i++){                //图节点

28         Node*node;
29         if(varList[i]->getArray()||varList[i]->inMem)
30             node=new Node(varList[i],U);
31         else
32             node=new Node(varList[i],E);
```

```

33         varList[i]->index=i;
34         nodes.push_back(node);
35     }
36     Set buf=liveE;
37     list<InterInst*>::reverse_iterator i;
38     for(i=optCode.rbegin();
39         i!=optCode.rend();++i){
//冲突边

40         Set& liveout=(*i)->liveInfo.out;
41         if(liveout!=buf){
42             buf=liveout;
43             vector<Var*> coVar=lv->getCoVar(liveout & mask);
44             for(int j=0;j<(int)coVar.size()-1;j++){
45                 for(int k=j+1;k<coVar.size();k++){
46                     nodes[coVar[j]->index]->
47                         addLink(nodes[coVar[k]->index]);
48                     nodes[coVar[k]->index]->
49                         addLink(nodes[coVar[j]->index]);
50                 }
51             }
52         }
53     }
54 }

```

第5行记录变量活跃性信息，第6~7行根据可用寄存器个数`regNum`初始化颜色集合全集`U`和颜色集合空集`E`。

第8~22行将参数变量和局部变量添加到变量列表（全局变量不进行寄存器分配），并将取址运算的操作数变量标记为必须在内存而不能分配寄存器，即`inMem`为`true`。

第23~26行计算掩码集合`mask`。变量活跃性分析时的变量集合包含全局变量，而当前变量集合不包含全局变量，因此将`mask`对应全局变量的索引元素设置为0，其他变量对应的索引元素设置为1。当从变量活跃性信息`lv`中取出活跃变量集合`out`时，需要将`out`与`mask`进行与运算，仅保留`out`集合中与非全局变量相关的活跃变量信息。

第27~35行构建冲突图节点，第29~30行为数组或标记为inMem的变量创建图节点，并将节点的exColor集合设为全集U，即该节点不能使用任何颜色。第32行为一般的变量创建图节点，并将exColor设计为空集E，表示寄存器分配前，节点可以使用任何原色。第34行将创建的图节点保存到节点列表nodes。

第36~53行为冲突图添加关联边，第36~42行逆序遍历所有的指令，并取出指令出口处的活跃变量信息liveout。变量buf用于缓存上一条指令的liveout，避免连续等值的liveout的重复计算。

第43行调用getCoVar获取liveout对应的所有变量集合coVar，mask用于过滤全局变量。

第44~51行将covar的变量两两组合，调用addLink添加冲突边。

```
1 void Node::addLink
(Node*node){
2     vector<Node*>::iterator pos=
3         lower_bound(links.begin(),links.end(),node);
4     if(pos==links.end() || *pos!=node){
5         links.insert(pos,node);
6         degree++;
7     }
8 }
```

函数addLink将当前节点添加到节点node的冲突边。lower_bound函数根据二分查找算法将node节点按序插入到links。第6行将节点的度加1。

冲突图构造完毕后，调用图着色算法为图节点分配颜色。

```
1 void CoGraph::regAlloc
() {
2     Set colorBox=U;                                //颜色集合

3     int nodeNum=nodes.size();                      //节点个数

4     for(int i=0;i<nodeNum;i++){
5         Node* node=pickNode();                      //选取最大度节点

6         node->paint(colorBox);                      //对节点着色

7     }
8 }
```

第2行将colorBox初始化为颜色集合全集，表示所有的颜色。由于每次着色都会处理一个节点，因此图着色算法需要循环节点个数nodeNum次。pickNode函数用于选取冲突图中度最大的节点，paint函数为该节点进行着色。

pickNode函数的实现为：

```
1 Node* CoGraph::pickNode
() {
2     make_heap(nodes.begin(), nodes.end(), node_less());
3     Node* node=nodes.front();
4     return node;
5 }
```

函数pickNode调用make_heap函数将图节点序列构造为最大堆，那么图节点序列的第一个元素nodes.front（）便是最大度节点。

paint函数的实现为：

```

1 void Node::paint
  (Set& colorBox){
2     Set availColors=colorBox-exColors;                                //可用颜色集合

3     for(int i=0;i<availColors.count;i++){
4         if(availColors.get(i)){
5             color=i;
6             var->regId=color;
7             //着色

8             degree=-1;
9             //着色成功

10            for(int j=0;j<links.size();j++)                            //关联节
11                links[j]->
12                addExColor(color);
13            return ;
14        }
15    }
16    degree=-1;
17    //着色失败

18 }
19 void Node::addExColor
  (int color){
20     if(degree==-1)return;                                            //已
21     //经着色

22     exColors.set(color);                                            //添加
23     //排除色

24     degree--;
25     //节点的度减

26 }
27 }

```

第2行计算节点可以使用的颜色集合`availColors`，第4行表示还有颜色可用，将颜色编号`i`保存到变量的`regId`字段，表示为变量分配的寄存器编号。

第7行将**degree**设置为最小值-1，这样该节点就不会成为度最大的节点。

第8~9行处理关联的节点，调用**addExColor**为关关节点添加不可使用颜色。

第13行表示节点没有可用颜色，无法分配寄存器，将**degree**设置为-1不再处理。

第17行将节点不可用颜色添加到**exColor**字段，并将节点的度-1，相当于删除了该节点与被着色节点的关联边。

图着色算法执行完成后，每个变量（全局变量除外）的**regId**字段都保存了变量被分配到的寄存器编号。如果该字段为-1，则表示变量未分配到寄存器。

4.3.2 变量栈帧偏移计算

在前面对符号表管理的描述中，每次向符号表添加一个局部变量时，都会调用Fun对象的locate函数为变量计算相对栈帧基址的偏移，以保证局部变量的正常访问。然而，寄存器分配后，部分变量被保存到寄存器中而不再占用内存。因此，我们只需要为不能分配到寄存器的局部变量分配内存即可，这样可以减少栈帧空间的浪费。

如图4-11所示，原栈帧内需要保存局部变量a、b、c、d，经过寄存器分配后，变量a分配到寄存器eax，变量c分配到寄存器ebx，而变量b、d仍保存在内存中。将栈帧紧凑后，原变量a、c的空间被删除，而变量b、d的内存空间向栈帧基址处紧凑。栈帧紧凑后，变量b、d相对于栈帧基址的偏移发生了变化。因此寄存器分配后，如果要对栈帧内存进行紧凑，则必须重新计算局部变量的栈帧偏移。

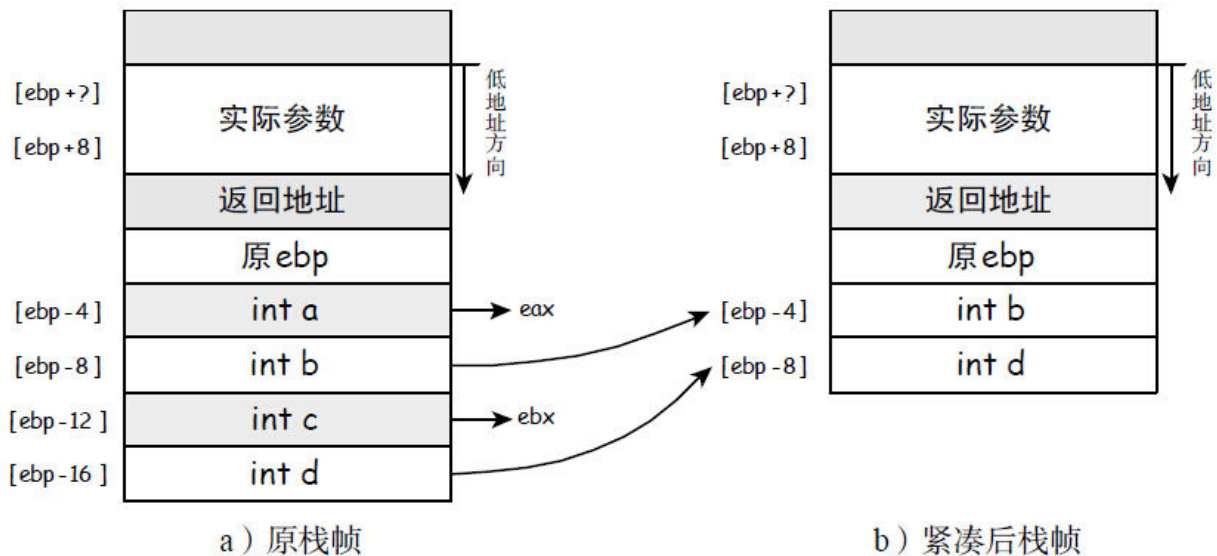


图4-11 局部变量紧凑

寄存器分配后，变量的`regId`字段记录了被分配到的寄存器编号，而`regId=-1`的变量仍需保存在栈帧。另外，中间代码的`OP_DEC`指令记录了局部变量的声明顺序。根据这些信息，我们确定了保留在栈帧内的所有局部变量。但是不同作用域的变量可能拥有同样的栈帧偏移量，比如代码：

```

if(x){
    int a;
}else{
    int b;
}

```

根据符号表中对作用域管理的描述，`if`分支作用域和`else`分支作用域拥有相等的作用域基址，故而变量`a`和`b`的栈帧偏移地址相同。所以重新计算变量栈帧偏移量时，需要明确变量所在的作用域。在变量`Var`对象内，`scopePath`字段保存了变量的作用域路径，即全局作用域到变

量所在作用域的路径。根据局部变量的作用域路径，可以完全还原代码的作用域嵌套结构。例如代码：

```
//全局作用域

0
void fun(){                                     //函数作用域
域

1
    int a;                                     //定义
//定义

a
    if(a){int b;}                             //if作用域

2. 定义

b
    else {
//else作用域

3
        int c;                               //定义
//定义

c
        while(a){int d;}                     //while作用域

4. 定义

d
    }
//定义
int e[10];                                     //定义

e
}
```

所有局部变量的作用域路径为：

```
PATH(a, b, c, d, e)={/0/1, /0/1/2, /0/1/3, /0/1/3/4, /0/1}。
```

我们可以根据如下栈帧偏移算法构建作用域树，并求解变量的栈帧偏移量。

- 1) 创建根作用域节点0，其栈帧偏移初始化为0。
- 2) 取下一个dec指令保存的未分配寄存器的局部变量，获得其作用域路径。
- 3) 从左到右解析作用域路径，并从树根处开始自顶向下进行匹配。如果作用域节点不存在则创建作用域节点，新创建的作用域节点初始化为父节点的栈帧偏移。
- 4) 取出作用域路径匹配结束时的作用域节点，将变量的大小累加到作用域节点的栈帧偏移，并将新的栈帧偏移设置为变量的栈帧偏移。
- 5) 跳转到2)处，直到所有的指令处理完毕。

假定前面示例代码的变量都未分配到寄存器，则使用栈帧偏移算法构造作用域树，算法执行流程如图4-12所示。图中第1步首先将根的作用域初始化为0，其作用域栈帧偏移初始值为0，当前值为0。第2步处理局部变量a的声明，取出作用域路径“/0/1”，与作用域树匹配时，不存在节点1，因此创建节点1，其栈帧偏移初始值为父节点栈帧偏移的当前值，然后将变量a的大小4累加到作用域1的栈帧偏移，因此得到变量a的栈帧偏移为4。类似地，得到变量b的栈帧偏移为8，变量c的栈帧偏移为8，变量d的栈帧偏移为12。处理局部数组e的声明时，取出作

用域路径“/0/1”，与作用域树匹配得到节点1，作用域节点1的当前栈帧偏移为4，累加变量e的大小后得到数组e的栈帧偏移为44。

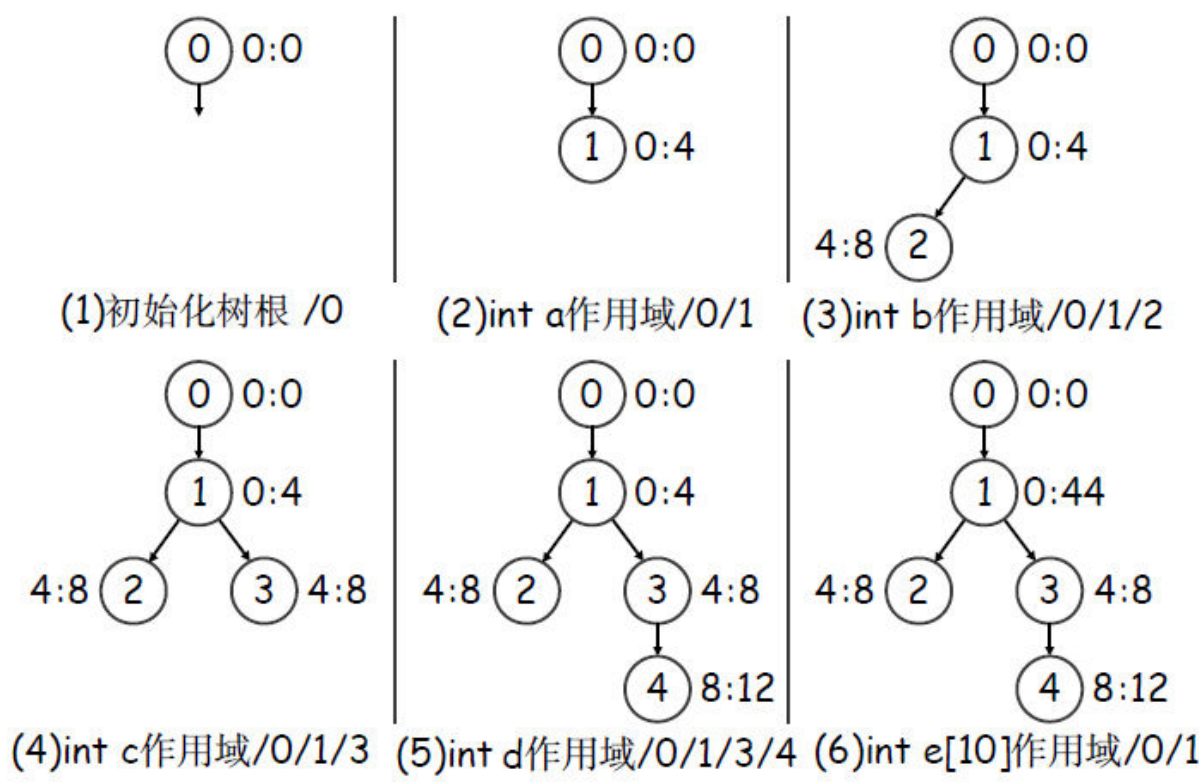


图4-12 栈帧偏移计算执行流程图

作用域树相关的数据结构定义为：

```
1 struct Scope
2 {
3     struct scope_less{
4         bool operator()(Scope*left,Scope*right){
5             return left->id<right->id;
6         }
7     };
8     int id;
9     //作用域编号

10    int esp;
11    //栈帧偏移

12    vector<Scope*> children;
13    //子作用域
```

```

10     Scope*parent;
    //父节点

11 };
12 class CoGraph

{
13     Scope* scRoot;
    //根作用域

14 };

```

类型Scope表示作用域树的节点。函数对象scope_less用于比较两个作用域节点的大小，字段id记录作用域的编号，esp记录作用域的栈帧偏移，children记录作用域的子作用域序列，parent记录父作用域节点。冲突图类型CoGraph内的scRoot记录了作用域树的树根。

根据作用域树数据结构，栈帧偏移计算算法实现如下：

```

1  void CoGraph::stackAlloc

(){
2      scRoot=new Scope(0,0);
    //根作用域

3      int max=0;
4      for(list<InterInst*>::iterator i=optCode.begin();
5          i!=optCode.end();++i){
6          InterInst*inst=*i;
7          Operator op=inst->

getOp();
8          if(op==OP_DEC){
    //局部变量

9              Var* arg1=inst->

getArg1();
10             if(arg1->regId==-1){
    //没有分配到寄存器

11                 int& esp=getEsp(arg1->getPath());
    //作

    esp

```

```

12                                     int size=arg1->getSize();
//变量大小

13                                     size+=(4-size%4)%4;
//4字节对齐

14                                     esp+=size;
//修改

esp
15                                     arg1->setOffset(-esp);
//记录偏移

16                                     if(esp>max)max=esp;
17                                     }
18                                     }
19     }
20     fun->setMaxDep(max);
21 }

```

第2行创建根作用域节点，即全局作用域0，第3行的max变量记录栈帧紧凑后的栈帧大小。

第4~9行遍历中间代码，并取出局部变量声明指令OP_DEC和局部变量arg1。

第10行判断变量的regId如果等于-1，表示变量没有分配到寄存器，需要计算栈帧偏移。

第11行调用getEsp函数取出变量所在作用域节点的esp变量。

第12~14行将变量大小按照4字节对齐后，累计到作用域栈帧偏移esp。

第15行将值-esp设置为变量的栈帧偏移，第16行计算栈帧的大小，第20行将紧凑后的栈帧大小保存到函数对象。

函数getEsp根据变量的作用域路径查询或构建作用域树。

```
1  int& CoGraph::getEsp
   (vector<int>& path){
2      Scope* scope=scRoot;
3      for(unsigned int i=1;i<path.size();i++)           //查找作用域

4          scope=scope->find(path[i]);
5      return scope->esp;
   //返回作用域

   esp
6  }
7  Scope* Scope::find
   (int i){
8      Scope*sc=new Scope(i,esp);                       //创
   建子作用域

9      vector<Scope*>::iterator pos=lower_bound
10         (children.begin(),children.end(),sc,scope_less());
11      if(pos==children.end() || (*pos)->id!=i){
12          children.insert(pos,sc);
   //插入作用域

13          sc->parent=this;
   //记录父节点

14      }
15      else{
16          delete sc;
17          sc=*pos;
   //找到作用域

18      }
19      return sc;
20 }
```

第3~4行从左向右处理作用域路径path的每个作用域编号，并以此为参数调用find查询作用域节点，记录到scope。作用域路径遍历结束后，scope内保存了作用域路径path对应的作用域对象。

函数find根据编号查找当前作用域的子作用域。第8行创建查询节点sc，第9~10行使用二分查找算法查询作用域编号为参数i的作用域节

点，比较函数为`scope_less`。

第12~13行表示为查询到编号为*i*的作用域节点，将编号*i*的作用域节点插入到`children`序列，并记录当前作用域节点到新创建的子作用域的`parent`字段。

第16~17行表示找到了编号为*i*的作用域节点，于是将查询节点删除，并返回查找到的作用域节点。

通过图着色算法和栈帧偏移计算，完成了变量的寄存器分配和栈帧内存的紧凑，为生成更高效的目标代码提供了可能。不过本节设计的寄存器分配方案只是一种粗略的实现，与现代编译器的寄存器分配还有很大差距，仅说明了寄存器分配的主要思想。在目标代码生成阶段，需要考虑的问题还有很多。

对于CISC指令集，通用寄存器的个数十分有限。在32位的x86指令集中，通用寄存器只有8个，除去寄存器`ebp`和`esp`用于函数栈帧管理不能用于存放操作数和计算结果外，目标代码生成时还使用了额外的通用寄存器用于缓存操作数和计算结果，因此留给寄存器分配的寄存器资源就更加紧张了。如果使用前面描述的目标代码生成算法，本节实现的寄存器分配算法可能更适用于RISC指令集，因为RISC指令集提供了更多的通用寄存器。

不同的函数将寄存器分配给自身的变量，那么在函数调用前后需要对通用寄存器进行保存和恢复操作，以避免寄存器数据的混乱。另外，我们为函数的参数分配了寄存器，为了保证函数对参数访问时，参数值已经保存在寄存器，必须在函数最开始执行时将参数加载到对应的寄存器。

4.4 窥孔优化

目标代码生成阶段，产生的汇编代码并非足够简洁，其中可能存在可以化简的指令序列。比如表达式“ $a=b+c$ ；”，假设变量 a 、 b 、 c 都是全局`int`变量，那么生成的汇编代码可能为：

```
1  mov  eax,[b]
2  mov  ebx,[c]
3  add  eax,ebx
4  mov  [c],eax
```

我们希望最终的汇编代码形式为：

```
mov  eax,[b]
add  eax,[c]
mov  [c],eax
```

这是因为`x86`指令集提供了操作数为寄存器和内存的`add`指令，因此可以将指令2和3合并为一条指令。我们可以借鉴窥孔优化的思想，实现对汇编代码的优化。

窥孔优化器对目标代码线性扫描，使用一个固定大小的滑动窗口监视扫描位置的代码序列，并将该序列与已设定的代码模板进行匹配，执行指令的替换、消除、合并等优化动作。如图4-13所示，滑动窗口（图中黑色区域）发现局部代码序列“`mov ebx, [c]`”和“`add eax, ebx`”与已有的代码模板匹配，因此使用对应的代码简化规则将其化简

为“add eax, [c]”。然后窗口继续向后滑动，移入后续的指令，重复以上过程。

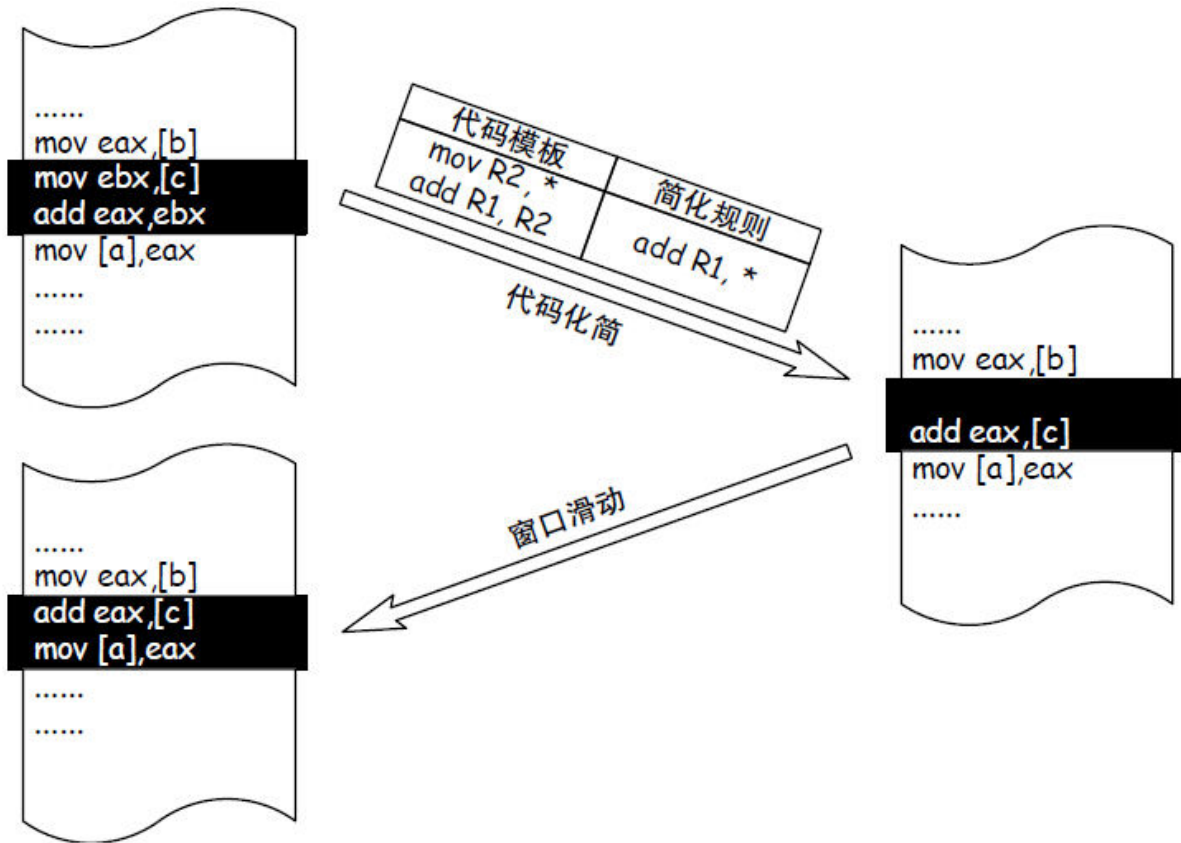


图4-13 窥孔优化

为了简化问题的讨论，我们假定滑动窗口的大小为2，且代码模板匹配成功后，使用一条指令替换滑动窗口的代码序列。

窥孔优化涉及的数据结构定义如下。

```
1 class Window
{
2     list<Asm*> cont; //窗口内的指令

3     list<Asm*>& code; //目标代码序列
```

```

4         list<Asm*>::iterator pos;                                //窗口位置

5         void replace(Asm* inst);                                //代码化简

6     public:
7         bool move();                                            //窗口移动函数

8         bool match();                                            //指令模板匹配

9     };
10    class PeepHole
11    {
12        list<Asm*>& code;                                        //目标代码序列

13    public:
14        void filter();                                            //窥孔优化
15    };

```

滑动窗口类**Window**的定义中，字段**cont**表示被滑动窗口“监视”的指令序列，**code**为目标代码序列，**pos**字段表示滑动窗口在目标代码的位置。函数**move**用于向后移动滑动窗口，**match**将窗口内的指令序列与代码模板匹配，并调用**replace**函数对代码进行化简。

窥孔优化类**PeepHole**的定义中，字段**code**表示目标代码序列，函数**filter**进行窥孔优化操作。

类**Window**的**move**函数实现滑动窗口向后移动，实现代码为：

```

1    bool Window::move
2    () {
3        for(pos==code.end()){
4            Arm* inst=*pos;
5            cont.pop_front();
6            cont.push_back(inst);
7            pos++;
8            return true;
9        }
10   }

```

```
8     }
9     return false;
10 }
```

由于限定滑动窗口的大小为2，且只使用一条指令执行化简操作。因此窗口滑动时，只需要将窗口内移入一条后继指令即可。

第4~5行将cont的首元素弹出，然后压入后继指令。第6行累加滑动窗口的位置。

类Window的match函数实现代码模板的匹配，实现代码为：

```
1 void Window::match
2 (){
3     Asm& inst1=**cont.front();           //取出指令
4
5     Asm& inst2=**cont.back();           //取出指令
6
7     AsmOp op1=inst1.op;
8     AsmArg a11=inst1.arg1;
9     AsmArg a12=inst1.arg2;
10    AsmOp op2=inst2.op;
11    AsmArg a21=inst2.arg1;
12    AsmArg a22=inst2.arg2;
13    if(op1.isMov()&&op2.isAdd()&&
14        a11.isReg()&&a21.isReg()&&a22.isReg()&&a11==a22){
15        replace(new Asm("add",a21,a12));
16    }
17    else if                               //其他指令模板
18
19
20
21 }
```

第2~9行，match函数将滑动窗口内的指令的内容取出。

第10~12行执行代码模板匹配，该代码模板来源于图4-13的例子。即判断如果指令1是mov指令，指令2是add指令，指令1的第一个操作数和指令2的第二个操作数都是寄存器且为同一个寄存器，指令2的第一个操作数也是寄存器，那么将这两条指令化简为一条add指令，操作数1为指令2的第一个操作数，操作数2为指令1的第二个操作数。

代码化简由replace实现，第18~19行描述了化简的过程。由于滑动窗口的大小为2，因此只需要将第一条指令设置为死指令，将第二条指令的内容用新指令替换。这样当窗口移动后，标记为死的指令从滑动窗口中移出，被替换的指令可以与后继指令形成新的组合再与代码模板匹配。

上述代码只给出了图4-13描述的代码模板的匹配实现，我们可以根据实际需要添加新的代码模板和对应的简化规则，这体现了窥孔优化算法的灵活性。

根据滑动窗口的move和match函数，实现窥孔优化的代码为：

```
1 void PeepHole::filter
   (){
2     Window win(code);
3     bool flag=false;                                     //记录匹配
   成功出现标记

4     do{
5         win.match();                                     //执
   行匹配

6     }while(win.move());                                 //移
   动滑动窗口
```

```
7      list<Asm*>::iterator i=code.begin(),k=i;
8      for(;i!=code.end();i=k){                                //清除死
指令
9          k++;
10         if((*i)->isDead())
11             code.remove(i);
12     }
```

第2行创建滑动窗口win，第4~6行通过循环调用move函数将滑动窗口向后移动，并且每次窗口滑动后，都会调用match函数与代码模板库进行匹配，执行替换等操作进行目标代码优化。

第8~12行对目标代码进行清洗，将死指令从目标代码中删除。

4.5 本章小结

本章通过对中间代码优化和目标代码优化的若干经典优化算法的实现，描述了编译优化器的实现方式。我们从数据流分析框架开始，讨论了中间代码优化算法的实现。常量传播将变量的初始值传递到表达式计算指令中，复写传播降低了变量之间的关联程度，从而为死代码消除提供更多的可能，死代码消除根据变量的活跃性分析消除对程序结果无影响的指令。寄存器分配的图着色算法也使用了变量的活跃性信息数据流，同时为了紧凑栈帧内存，讨论了变量栈帧偏移的计算方法。最后，我们使用窥孔优化的方式对目标代码进行了优化。至此，我们完成了编译优化器的所有实现。

第5章 二进制表示

工欲善其事，必先利其器。

——《论语》

经过编译器的处理，高级语言程序被转化为目标机器的汇编代码。根据编译系统的流程，下一步便是将汇编代码转化为目标机器的二进制指令。鉴于汇编器的实现过程中，牵涉了大量机器指令和目标文件格式的内容，因此，在描述汇编器的实现前，需要对此有清晰的了解。

在编译器的构造阶段，除了代码生成阶段要考虑程序的运行时存储，我们不需要关心太多底层的细节。而在汇编器和链接器的构造阶段，必须清楚了解二进制代码和二进制文件的细节。我们的编译器产生的是Intel x86汇编程序，而编译系统生成的二进制文件是Linux系统下的ELF文件格式的文件。因此，本章讨论的主题是x86指令格式和ELF文件格式。

5.1 x86指令

要了解Intel x86指令格式的细节，最好的参考资料莫过于Intel指令开发手册，不过上千页的开发手册令人难以抓到重点。我们构造编译系统的目的的一方面是透析编译的细节和流程，另一方面也试图深入了解Intel的体系结构和指令格式的细节。当然，本书参考了Intel指令开发手册的部分内容，尽可能将指令的细节展示出来。

图5-1描述了x86指令的通用格式。一般的x86指令包含6个部分：指令前缀（Instruction Prefix）、操作码（Opcode）、ModR/M字段、SIB字段、偏移（Displacement）和立即数（Immediate）。

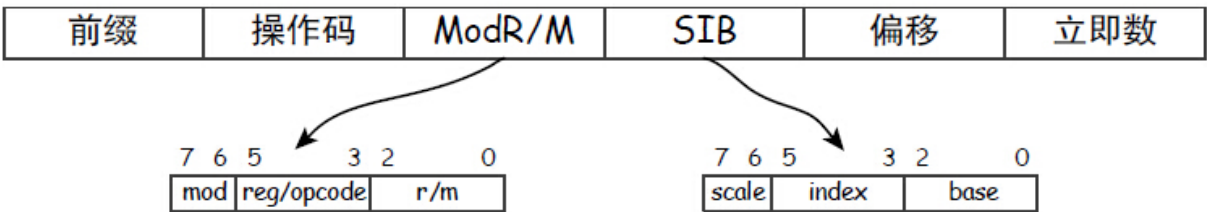


图5-1 x86指令格式

指令中一定会包含操作码字段，它表示指令的功能，而其他字段都是可选的。x86采用不定长指令编码，正常的x86指令长度最短为1个字节，最长为15个字节。指令前缀为指令提供附加的功能，ModR/M和

SIB字段一般提供操作数的访问模式，偏移和立即数字段直接编码到指令内部。下面对这些字段的细节逐一进行说明。

5.1.1 指令前缀

在前面讨论的编译器实现中，并未生成包含指令前缀的汇编指令。指令前缀一般用于增强指令功能，调整内存操作数访问属性等。

指令前缀可以分为如下几类：

1) **操作数大小重写前缀**。二进制编码为0x66。操作数大小重写发生在指令操作数大小与当前汇编上下文默认操作数大小不一致的情况。例如32位汇编语言指令：

```
mov eax,1
```

其指令编码为：

```
b8 01 00 00 00
```

而在16位汇编语言中，不能直接访问寄存器**eax**，因此上述指令编码的实际含义为：

```
mov ax,1
```

如果16位汇编语言要访问32位寄存器**eax**，必须使用前缀修改操作数大小：

66

b8 01 00 00 00

通过加入操作数大小重写前缀，达到“强制”访问32位寄存器eax的目的。类似的情况也发生在32位汇编语言访问16位寄存器的情况。

2) **地址大小重写前缀**。二进制编码为0x67。地址大小重写发生在指令内存操作数地址大小与当前汇编上下文默认地址大小不一致的情况。例如32位汇编语言指令：

```
mov eax, [0x12345678]
```

其指令编码为：

a1 78 56 34 12

而在16位汇编语言中，不能直接访问32位地址，因此上述指令编码内的地址被截断：

a1 78 56

实际汇编指令形式为：

```
mov ax, [0x5678]
```

如果16位汇编语言要访问32位地址，必须使用前缀修改地址大小：

67
a1 78 56 34 12

通过加入地址大小重写前缀，达到“强制”访问32位地址的目的。
类似的情况也发生在32位汇编语言访问16位地址的情况。

3) **段重写前缀**。如果要修改指令内存操作数的段寄存器，则需要使用段重写前缀。例如指令：

mov eax,[ebx]

该指令使用ebx寄存器间址访问内存，默认当前段寄存器为数据段寄存器ds。如果需要指定段寄存器为es，那么指令形式为：

mov eax,es
: [ebx]

对应的指令编码为：

26
89 03

其中0x26表示段寄存器es对应的指令前缀。不同段寄存器对应指令前缀的二进制编码见表5-1。

表5-1 段重写前缀

段寄存器	cs	ds	es	fs	gs	ss
指令前缀	0x2e	0x3e	0x26	0x64	0x65	0x36

在32位内存模式下，对内存操作数的段重写已经失去意义。无论是为内存操作数指定段寄存器还是指定不同的段寄存器，都不会影响指令的功能。

4) **重复执行指令前缀**。包括rep/repz和repnz指令前缀，对应的二进制编码分别为0xf3和0xf2。该类指令一般用于串操作指令，例如：

```
rep
movsb
```

指令编码为：

```
f3
a4
```

5) **lock前缀**。二进制编码为0xf0。该指令用于指令执行时锁定地址总线，保证对称多处理器（SMP）环境下对内存数据访问的原子性。例如：

```
lock
add eax, [ebx]
```

指令编码为：

f0
01 03

6) **分支提示前缀**。分支提示前缀仅用于条件跳转指令Jcc，表示条件跳转在大多数情况下是否发生。前缀ht表示Jcc指令大多数情况跳转成功，二进制编码为0x3e，前缀hnt表示Jcc指令大多数情况不发生跳转，二进制编码为0x2e。例如指令：

hnt
jne L
L:

该指令表示jne指令大多数情况不会跳转到L，其指令编码为：

2e
0f 85 00 00 00 00

以上讨论了传统的x86指令前缀，在AMD推出x86扩展64位技术之后，增加了新的用于访问64位数据的指令前缀（**REX prefix**），而原本的x86指令前缀称为原始前缀（**Legacy prefix**）。因此64位汇编指令可以使用这两类指令前缀，而32位汇编指令仅能使用**Legacy**前缀。由于我们只关心32位汇编指令的内容，因此不再对**REX**前缀进行深入讨论。

5.1.2 操作码

指令的操作码表达了指令的基本功能，是指令中必不可少的字段，其长度为1~3字节不等。1字节操作码与指令前缀共享指令第一个字节的空间（0x00—0xff），这是因为指令前缀是可选的，CPU的解码器根据指令第一个字节的值确定该字节是指令前缀还是操作码。比如遇到指令第一个字节为0x66，则为操作数大小重写前缀。如果遇到0xb8，则为mov指令的一种操作码。2字节操作码总是以0x0f开始，紧跟另一个字节。其中0x0f字节称为操作码的转义前缀（Escape Prefix）或转义操作码。例如jne指令的操作码为“0f 85”。与2字节操作码类似，3字节操作码也是使用转义前缀的方式进行扩展编码。3字节操作码的转义前缀包含“0f 38”和“0f 3a”两种，转义前缀后紧跟另一个字节共同表示操作码。

我们设计的编译器生成的汇编指令的操作码长度都是2字节以内，因此我们只讨论1字节和2字节的操作码。根据操作码表达指令功能的不同方式，将常见的指令操作码分为四类分别讨论。

1) **操作码独立表示指令功能**。对于1字节长度指令，其操作码表达了指令的所有含义。如表5-2中指令ret二进制编码为0xcb。类似的1字节编码的指令比较少，比如nop、leave、int 3等。

表5-2 操作码表（一）

指 令	操作码 (0x)	举 例
ret	cb	ret
nop	90	nop
leave	c9	leave
int 3	cc	int 3
jne rel32	0f 85	jne L
jmp rel32	e9	jmp L
call rel32	e8	call fun
int imm8	cd	int 0x80
push imm32	68	push 0x12345

更常见的是操作码后附加操作数的指令。比如前面讨论过的条件跳转指令（Jcc）的jne指令，便是操作码“0f 85”后紧跟4字节的目标标签的相对地址。类似的指令包括Jcc、jmp、call、int、push，其中rel32表示32位相对地址，imm8表示8位立即数，imm32表示32位立即数。

仅使用操作码就能表达功能的指令，其操作数形式都比较简单。当操作数访问模式复杂时，则需要使用ModR/M和SIB字段补充指令的功能。Intel指令系统将操作数寻址模式相同的一类指令组成一组，使用共同的操作码表示，这样的操作码称为组属性操作码。我们根据不同的指令对操作码补充方式的不同，将操作码又分为以下（2）、（3）、（4）所述的三类。

2) 组属性操作码，ModR/M和SIB字段仅指定操作数访问模式。这类操作码仅定义了指令的基本框架，并未明确具体的操作数，具体的操作数由ModR/M和SIB字段给出。

如表5-3所示，其中r32表示32位寄存器，r/m32表示32位寄存器或内存操作数。例如指令“mov r32, r/m32”的操作码0x8b仅表示该指令可以从32位寄存器或内存中取出数据保存到32位寄存器中，但并未说明具体是哪个寄存器和内存地址。而ModR/M和SIB字段提供了这样的信息，这两个字段的具体细节后面会详细解释。类似的指令还有add、sub、cmp、lea、SETcc指令等。

表5-3 操作码表（二）

指 令	操作码 (0x)	举 例
mov r32,r/m32	8b	mov eax,ebx
mov r/m32,r32	89	mov [eax],ebx
add r32,r/m32	03	add eax,ebx
add r/m32,r32	01	add [eax],ebx
sub r32,r/m32	2b	sub eax,ebx
sub r/m32,r32	29	sub [eax],ebx
cmp r32,r/m32	3b	cmp eax,ebx
cmp r/m32,r32	39	cmp [eax],ebx
lea r32,r/m32	8d	lea eax,[ebx]
sete r8	0f 94	sete al

3) 组属性操作码，ModR/M的reg字段对操作码作补充。前面讨论的ModR/M字段仅提供了指令的操作数访问模式信息，除此之外，ModR/M的reg字段还具有对组属性操作码进行补充的功能，即反作用于操作码。

ModR/M字节的3~5位表示reg字段，取值范围为0~7，该字段可以对组属性的操作码进行补充。表5-4操作码列中的“/”符号后的数字表示

reg字段的值，例如imul和idiv指令的操作码都是0xf7，当reg=5时表示imul指令，当reg=7时表示idiv指令。

表5-4 操作码表（三）

指 令	操作码 (0x)	举 例
imul r/m32	f7 /5	imul ebx
idiv r/m32	f7 /7	idiv ebx
neg r/m8	f6 /3	neg al

(续)

指 令	操作码 (0x)	举 例
neg r/m32	f7 /3	neg eax
inc r/m8	fe /0	inc al
dec r/m8	fe /1	dec al
add r32,imm32	81 /0	add eax,0x12345
sub r32,imm32	81 /5	sub eax,0x12345
cmp r32,imm32	81 /7	cmp eax,0x12345

4) 组属性操作码，寄存器编号补充操作码。除了ModR/M的reg字段可以对操作码进行补充外，寄存器本身的编号也可以对操作码进行补充。

如表5-5所示，inc指令的组属性操作码为0x40，当指定了具体的操作数寄存器后，需要将该寄存器的编号累加到操作码中，类似的指令还有dec、push、pop、mov等。

表5-5 操作码表（四）

指 令	操作码 (0x)	举 例
inc r32	40+reg	inc eax
dec r32	48+reg	dec eax
push r32	50+reg	push eax
pop r32	58+reg	pop eax
mov r32,imm32	b8+reg	mov eax,0x12345

Intel指令集中的寄存器编号都是固定的，常见的寄存器编号的值如表5-6所示。

表5-6 寄存器编号

寄存器编号	0	1	2	3	4	5	6	7
8 位寄存器	al	cl	dl	bl	ah	ch	dh	bh
16 位寄存器	ax	cx	dx	bx	sp	bp	si	di
32 位寄存器	eax	ecx	edx	ebx	esp	ebp	esi	edi

我们发现通用寄存器的eax、ebx、ecx、edx的编号并不是按照名字递增的顺序编号，而是被分别编号为0、3、1、2，这点需要注意。

5.1.3 ModR/M字段

在讨论操作码的内容时，我们涉及了ModR/M字段（见图5-2）对操作码补充或标识操作数访问模式的功能，下面详细讨论该字段的具体含义。

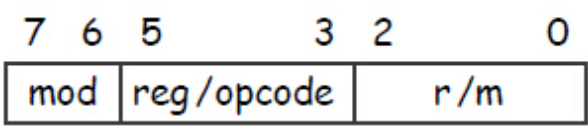


图5-2 ModR/M字段

ModR/M字段长度为1字节，其中0~2位为r/m字段、3~5位为reg字段、6~7位为mod字段。reg字段保存寄存器编号或操作码的补充信息。r/m字段表示另一个操作数，也保存寄存器编号，该编号指定的寄存器中可能是操作数本身，也可能是存放与操作数的内存地址相关的信息。mod字段为r/m字段指定具体的操作数模式，r/m字段内保存寄存器编号，根据不同mod的值确定该寄存器是寄存器操作数，还是需要寻址的内存操作数。如表5-7所示。

表5-7 mod字段含义

mod	r/m 寻址模式	举 例
00	寄存器间址	[eax]
01	寄存器基址+8 位偏移	[eax+4]
10	寄存器基址+32 位偏移	[eax+0x12345]
11	寄存器操作数	eax

当mod=0b11时，r/m字段表示寄存器操作数，保存了寄存器的编号。当mod取其他值时，r/m字段表示内存操作数，保存了通过寄存器寻址的寄存器编号。其中mod=0b00时，表示寄存器间址。mod=0b01时，表示寄存器基址+8位偏移的内存寻址。mod=0b10时，表示寄存器基址+32位偏移的内存寻址。

例如0x8b表示指令“mov r32, r/m32”的操作码，其中r32对应reg字段，r/m32对应r/m字段。由此指令“mov ecx, eax”的编码为：

8b c8

其中，0xc8为ModR/M字段的值，表示为二进制形式为：

11 001 000

可以看出mod=0b11，表示r/m字段为寄存器操作数。reg=0b001，表示mov指令的操作数r32=ecx。r/m=0b000，表示mov指令的另一个操作数r/m32=eax。因此，二进制编码“8b c8”表达了汇编指令“mov ecx, eax”的完整信息。

类似地，指令“mov ecx, [eax]”的ModR/M的mod字段为0b00，表示r/m字段为寄存器间址，其他字段不变，指令编码为：

8b 08

而对于指令“mov ecx, [eax+4]”的ModR/M的mod字段为0b01，表示r/m字段为寄存器基址+8位偏移寻址。其他字段不变，但是需要增加1字节的偏移字段0x04。指令编码为。

8b 48 04

类似地，指令“mov ecx, [eax+0x12345678]”的ModR/M的mod字段为0b10，表示r/m字段为寄存器基址+32位偏移寻址。其他字段不变，但是需要增加4字节的偏移字段0x12345678。指令编码为：

8b 88 78 56 34 12

这里需要注意的是，对于指令中编码的偏移或立即数，都是按照小端字节序（Little Endian）的方式存储的，即高字节数据存储在高位地址，低字节数据存储在低位地址。因此偏移“0x12345678”存储形式为“78563412”，而非“12345678”。

我们发现，以上讨论的mod字段对r/m字段定义的通用寻址模式中，仅包含三种寻址模式：寄存器寻址（寄存器操作数）、寄存器间址和寄存器基址+偏移的寻址方式。除此之外，在指令系统中还存在立即寻址（立即数操作数）、直接寻址（直接使用内存地址寻址）和寄存器基址+寄存器变址+偏移的寻址方式。

对于立即寻址，Intel x86指令集的立即数一般都是硬编码在指令内部，而不需要ModR/M字段指定。例如表5-5中的指令“mov r32, imm32”，其操作码为0xb8+reg，因此对于指令“mov ecx, 0x12345678”，其中reg保存ecx的寄存器编号0b001，其指令编码为：

```
b9 78 56 34 12
```

对于直接寻址，Intel x86指令集规定，当mod=0b00，r/m=0b101时，表示32位直接寻址模式。例如指令“mov ecx, [0x12345678]”，其指令编码为：

```
8b 0d 78 56 34 12
```

在表5-6中，0b101是寄存器ebp的编号，由于该编号被直接寻址模式占用，因此形如“mov r32, [ebp]”的指令，必须转化为“mov r32, [ebp+0]”进行处理。例如指令“mov ecx, [ebp]”的指令编码为：

```
8b 4d 00
```

这样使用[ebp]寻址的指令需要额外使用1字节的存储。

对于寄存器基址+寄存器变址+偏移的寻址方式，仅使用ModR/M字段无法表示。Intel指令集规定，当mod!=0b11，r/m=0b100时，表示引

导SIB字段。由SIB字段表示ModR/M字段无法表示的寻址模式，而mod定义的偏移信息仍然有效。

类似地，由于引导SIB字段占用了寄存器编号0b100，对应寄存器esp。因此形如“mov r32, [esp]”“mov r32, [esp+disp8]”“mov r32, [esp+disp32]”的指令也无法仅使用ModR/M字段表示。

特殊ModR/M字段如表5-8所示。

表5-8 特殊ModR/M字段

mod	r/m	r/m 寻址模式	举 例
00	101	32 位直接寻址	[0x12345]
00	100	引导 SIB	[?]
01	100	引导 SIB+8 位偏移	[?+4]
10	100	引导 SIB+32 位偏移	[?+0x12345]

5.1.4 SIB字段

SIB字段为ModR/M字段补充寻址模式，如图5-3所示，通用寻址模式为寄存器基址+寄存器变址+偏移的寻址，当然也解决了上述由于寄存器编号冲突导致的部分指令无法由ModR/M字段表示的问题。

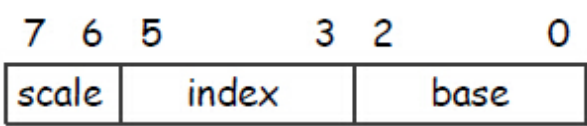


图5-3 SIB字段

SIB字段长度为1字节，其中0~2位为base字段、3~5位为index字段、6~7位为scale字段。base字段保存基址寄存器的编号，index字段保存变址寄存器的编号，scale字段保存以2为底的指数，表示变址寄存器的因子。因此，SIB字段定义的寻址模式格式为：

$$[base+index*2^{scale}]$$

例如指令“mov ecx, [eax+ebx]”中，内存操作数使用了变址寄存器ebx（当然，也可以将eax看作变址寄存器），因此必须使用SIB字段辅助寻址。指令编码为：

8b 0c 18

其中0x0c为ModR/M字段，二进制编码为00001100。mod=0b00表示指令不存在偏移，reg=0b001表示操作数ecx，r/m=0b100表示引导SIB字段。

0x18为SIB字段，二进制编码为00011000。scale=0b00表示变址寄存器因子为2⁰=1，index=0b011表示变址寄存器ebx，scale=0b000表示基址寄存器eax。因此，使用ModR/M和SIB字段完整表达了指令的功能。

类似地，对于指令“mov ecx, [eax+ebx*8+0x12345678]”，指令编码为：

```
8b 8c d8 78 56 34 12
```

其中ModR/M的字段mod=0b10表示指令中存在32位偏移，而SIB的字段scale=0b11，表示变址寄存器因子为2³=8，32位偏移仍按照小端字节顺序的方式存储。

SIB寻址模式见表5-9。

表5-9 SIB寻址模式

scale	寻址模式	举 例
00	[base+index]	[eax+ebx]
01	[base+index*2]	[eax+ebx*2]
10	[base+index*4]	[eax+ebx*4]
11	[base+index*8]	[eax+ebx*8]

前面讨论了SIB字段的通用寻址模式，但是当“基址+变址+偏移”的寻址方式中不存在基址寄存器或变址寄存器时，SIB需要进行特殊处理。

Intel指令集规定，当SIB的字段index=0b100时，不存在变址寄存器。这样SIB的寻址模式被简化为“[base]”形式，而使用ModR/M字段则可以完全表达该种的寻址模式。因此，不存在变址寄存器的寻址模式可以有两种不同的表达形式，例如指令“mov ecx, [eax]”的指令编码有数种：

(
1)
8b 08 (
2)
8b 0c 20 (
3)
8b 0c 60 (
4)
8b 0c a0 (
5)
8b 0c e0

第一种编码方式是仅使用ModR/M字段的情况，后四种编码方式是使用ModR/M和SIB字段的情况。使用SIB编码的情况中，必须设定index=0b100表示不存在变址寄存器，而scale字段可以取任意值0b00、0b01、0b10、0b11，base字段必须设置为0b000，表示基址寄存器eax。

之所以定义不存在变址寄存器的寻址模式，是为了表达在ModR/M字段的讨论中，由于寄存器编号冲突而无法表示的部分指令。例如指令“`mov ecx, [esp]`”的二进制编码为：

8b 0c 24

其中SIB字段的`base=0b100`表示基址寄存器`esp`，`index=0b100`表示不存在变址寄存器，`scale`可以取任意值，这里取值为`0b00`。

使用`index=0b100`表示不存在变址寄存器，导致一个很明显的结果——`esp`寄存器不能作为变址寄存器。事实确是如此，Intel汇编语法中不允许`esp`寄存器作为变址寄存器。例如指令“`mov ecx, [eax+esp*8]`”不是合法的指令。

接下来讨论不存在基址寄存器的情况，Intel指令集规定，当ModR/M的字段`mod=0b00`，SIB的字段`base=0b101`时，不存在基址寄存器，且寻址格式为：

$[index * 2^{scale}$
 $+ disp32]$

这里需要注意的是，不存在基址寄存器时，寻址模式中“强制”包含了32位的偏移。这是因为该寻址模式下，ModR/M的字段`mod=0b00`

并未指定指令包含偏移，因此这里将偏移字段补充进来。例如指令“`mov ecx, [eax*8+0x12345678]`”的编码为：

```
8b 0c c5 78 56 34 12
```

其中，SIB字段为0xc5，base=0b101表示无基址寄存器，但是必须有32位偏移，index=0b000表示变址寄存器为eax，scale=0b11表示变址寄存器因子为 $2^3=8$ 。

无基址寄存器的寻址模式也有一定的问题，例如当指令中不存在偏移时，指令“`mov ecx, [eax*8]`”，必须按照形式“`mov ecx, [eax*8+0x00000000]`”进行处理，指令编码为：

```
8b 0c c5 00 00 00 00
```

这样仅有变址寄存器的寻址模式的指令编码必须额外使用4字节的存储。

无基址寄存器寻址模式要求SIB的字段base=0b101，这和ebp的寄存器编号冲突。这样在mod=0b00时，ebp是无法作为基址寄存器的。但是，当mod=0b01或0b10时，ebp仍可以作为正常的基址寄存器使用。例如指令“`mov ecx, [ebp+eax*8+4]`”的指令编码为：

```
8b 4c c5 04
```

其中ModR/M的字段mod=0b01表示指令保存8位偏移，SIB的字段base=0b101表示基址寄存器ebp，index=0b000表示变址寄存器eax，scale=0b11表示变址寄存器因子 $2^3=8$ 。

而对于指令“mov ecx, [ebp+eax*8]”，虽然不存在偏移，但是需要按照“mov ecx, [ebp+eax*8+0]”的形式进行处理，指令编码为：

8b 4c c5 00

这样使用[ebp+index* 2^{scale}]寻址的指令需要额外使用1字节的存储。

5.1.5 偏移

32位Intel指令中的偏移分为两类：8位偏移和32位偏移。偏移配合ModR/M和SIB字段进行内存寻址，而不作为单独的字段出现在指令中。使用偏移进行寻址的方式包括：32位直接寻址、基址+8位偏移寻址、基址+32位偏移寻址、基址+变址+8位偏移寻址、基址+变址+32位偏移寻址和变址+32位偏移寻址，见表5-10。

表5-10 使用偏移的寻址模式

偏移	寻址模式	举 例
8 位偏移	基址+偏移	[eax+4]
	基址+变址+偏移	[eax+ebx*8+4]
32 位偏移	立即寻址	[0x12345]
	基址+偏移	[eax+0x12345]
	基址+变址+偏移	[eax+ebx*8+0x12345]
	变址+偏移	[ebx*8+0x12345]

对于8位偏移，一般由ModR/M的mod字段指定，mod=0b01时，表示指令使用8位偏移。对于32位偏移，也是由ModR/M的mod字段指定，mod=0b10时，表示指令使用32位偏移。另外需要注意的是，当ModR/M的mod=0b00且r/m=0b101时，表示指令使用32位偏移进行直接寻址。而当ModR/M的mod=0b00且SIB的base=0b101时，表示指令使用变址+32位偏移的寻址方式。

偏移在指令中如果存在，则紧跟ModR/M、SIB字段之后，且按照小端字节序的方式进行存储。

5.1.6 立即数

32位Intel指令中的立即数分为三类：8位立即数、16位立即数和32位立即数。不同长度的立即数操作数，操作码也不相同。例如指令“mov r8/16/32, imm8/16/32”的操作码见表5-11。

表5-11 不同长度立即数的mov指令操作码

立即数	指令前缀	操作码(0x)	举 例
8 位	无	b0+reg	mov cl,0x12
16 位	0x66	b8+reg	mov cx,0x1234
32 位	无	b8+reg	mov ecx,0x12345678

对于指令“mov r8, imm8”，指令的操作码为0xb0加上ModR/M的reg字段。例如指令“mov cl, 0x12”，指令编码为“b112”。

对于指令“mov r32, imm32”，指令的操作码为0xb8加上ModR/M的reg字段。例如指令“mov ecx, 0x12345678”，指令编码为“b978563412”。

对于指令“mov r16, imm16”，指令的操作码和32位立即数指令相同，但是在32位汇编语言环境下，需要添加操作数大小重写前缀0x66。例如指令“mov ax, 0x1234”，指令编码为“66 b93412”。

立即数在指令中如果存在，则紧跟ModR/M、SIB、偏移字段之后，且按照小端字节序的方式进行存储。

5.1.7 AT&T汇编格式

x86汇编语法有两种常见的格式：Intel汇编语法和AT&T汇编语法。Intel汇编语法常见于Intel官方文档和Windows平台，而AT&T汇编语法在Unix平台更为常见。一般使用NASM汇编Intel格式的汇编程序，而使用as汇编AT&T格式的汇编程序。

一般的汇编教材中，大多数使用Intel汇编语法，因此我们对Intel汇编语法更为熟悉，包括本书描述的汇编大都是该格式。但是在Linux环境中，无论是反汇编工具objdump，还是GNU C语言的嵌入式汇编，经常使用AT&T汇编格式。因此，我们有必要说明AT&T汇编语法的格式。通过对Intel汇编语法与AT&T汇编语法的比较，见表5-12，可以很容易理解它们的区别。

表5-12 操作数基本形式

	Intel 汇编语法	AT&T 汇编语法
寄存器	eax	%eax
立即数	0x1234	\$0x1234
操作数方向	mov eax,1	mov \$1,%eax

我们首先从汇编指令格式说明两种汇编语法的区别。

AT&T汇编的寄存器操作数前需要添加前缀“%”，立即数操作数需要添加前缀“\$”。两种汇编最大的不同是操作数位置相反，Intel汇编指

令形式为“助记符目的操作数，源操作数”，而AT&T汇编指令形式为“助记符源操作数，目的操作数”。

对于内存操作数，其一般的Intel汇编形式为“section: [base+index*scale+disp]”，而AT&T汇编形式为“%section: disp (%base, %index, scale) ”。

AT&T汇编经常涉及的内存操作数形式如表5-13所示。对于内存操作数，段寄存器section、基址寄存器base、变址寄存器index、变址寄存器因子scale、偏移disp都是可选的。当不存在基址寄存器base时，仍需要保留逗号分隔符。当不存在偏移disp时，默认为1。当仅有偏移disp时，表示直接寻址操作数，Intel汇编使用“[]”将内存地址包含起来，而AT&T直接使用内存地址进行访问。

表5-13 内存操作数

Intel 汇编语法	AT&T 汇编语法
[0x1234]	0x1234
[eax]	(%eax)
[eax*8]	(,%eax,8)

(续)

Intel 汇编语法	AT&T 汇编语法
[eax+0x1234]	0x1234 (%eax)
[eax+ecx]	(%eax,%ecx)
[eax+ecx*8]	(%eax,%ecx,8)
[ecx*8+0x1234]	0x1234 (,%ecx,8)
[eax+ecx*8+0x1234]	0x1234 (%eax,%ecx,8)
ds:[eax+ecx*8+0x1234]	%ds:0x1234 (%eax,%ecx,8)

由于内存操作数都是通过寻址的方式进行访问的，操作数的大小一般可以通过源寄存器或目的寄存器的大小自动推断。当内存操作数大小无法自动推断时（比如操作数中不存在寄存器时），必须显示指定操作数的大小，见表5-14。

表5-14 内存操作数大小

Intel 汇编语法	AT&T 汇编语法
mov byte ptr [eax],1	movb \$1, (%eax)
mov word ptr [eax],1	movw \$1, (%eax)
mov dword ptr [eax],1	movl \$1, (%eax)

Intel汇编使用“byte/word/dword ptr”前缀修饰内存操作数，表示操作数的大小是1、2、4字节。而AT&T则是通过在操作码后添加后缀“b/w/l”进行表示，分别对应单词“byte/word/long”。

一般情况下，内存操作数被作为数据对待，但是内存操作数被作为地址对待时，情况比较特殊，特殊内存操作数见表5-15。

表5-15 特殊内存操作数

Intel 汇编语法	AT&T 汇编语法
jmp dword ptr [0x1234]	jmp *0x1234
call dword ptr [eax]	call *(%eax)
jne dword ptr [eax+4]	jne *4(%eax)

当内存操作数作为跳转类指令（call、jmp、Jcc等）的目标地址时，需要使用前缀“*”修饰内存操作数。例如指令“jmp 0x1234”表示跳

转到地址0x1234处执行，而指令“`jmp*0x1234`”表示将地址0x1234处的内存数据取出，作为跳转地址。

接下来讨论两种汇编数据定义格式的区别，见表5-16。

表5-16 数据定义

Intel 汇编语法	AT&T 汇编语法
<code>ch db 'a'</code>	<code>ch: .byte 'a'</code>
<code>x dw 0x1234</code>	<code>x: .word 0x1234</code>
<code>var dd 0x12345678</code>	<code>var: .long 0x12345678</code>

(续)

Intel 汇编语法	AT&T 汇编语法
<code>array times 255 dd 0</code>	<code>array: .fill 255,4,0</code>
<code>str db "hello",0</code>	<code>str: .ascii "hello\000"</code>
<code>ptr dd str</code>	<code>ptr .long str</code>

Intel汇编使用“`db/dw/dd`”定义数据的长度，而AT&T使用“`.byte/.word/.long`”定义数据长度。Intel汇编使用“`times`”定义一块连续内存，而AT&T使用“`.fill`”定义连续内存。Intel汇编使用`db`后紧跟逗号分隔的常量列表定义字符串，而AT&T使用`.ascii`定义字符串。

除了以上所介绍的，两种汇编格式对应的汇编器执行指令也不同。Intel汇编程序使用`nasm`命令将汇编代码汇编为目标文件，命令格式为（`filename.s`为文件名）：

```
nasm -f elf filename.s
```

而AT&T汇编程序使用`as`将汇编代码汇编为目标文件。

```
as filename.s -  
o filename.o
```

5.2 ELF文件

目前主流的可执行文件格式有两种，Windows平台下的PE文件格式和Linux平台下的ELF文件格式。在Linux平台下，除了可执行文件（Executable File），可重定位目标文件（Relocatable Object File）、共享目标文件（Shared Object File）、核心转储文件（Core Dump File）也都是ELF格式的文件。

在我们设计的编译系统中，汇编器生成的目标文件是ELF格式的可重定位目标文件，链接器生成的是ELF格式的可执行文件。因此，详细了解ELF文件格式的细节，对构造汇编器和链接器至关重要。

文件头(ELF Header)	52
程序头表(Program Header Table)	(32*程序头表项个数)
代码段(. text)	
数据段(. data)	
bss 段(. bss)	0
段表字符串表(. shstrtab)	
段表(Section Header Table)	(40*段表项个数)
符号表(. symtab)	(16*符号表项个数)
字符串表(. strtab)	
重定位表(. rel . text)	(8*重定位表项个数)
重定位表(. rel . data)	(8*重定位表项个数)

图5-4 ELF文件结构

图5-4描述了ELF文件常见的结构（图中N表示表项的个数）。在ELF文件中，保存的最关键的信息是程序中的代码和数据。一般的，程序的代码以二进制指令的形式保存在代码段（.text）中，程序的数据以二进制的形式保存在数据段（.data）或“.bss”段中。ELF文件的其他结构，一般用于对ELF文件内容进行管理，为链接器、加载器、调试器、操作系统等提供必要的信息。

在Linux系统的“/usr/include/elf.h”头文件中，定义了ELF文件涉及的所有数据结构。根据ELF文件数据结构展开讨论ELF文件格式，更容易帮助我们把握ELF文件结构的细节。

后续对ELF文件结构补充说明的实例中，使用的可重定位目标文件file.o和可执行文件file由第1章示例中helloworld程序的源代码编译生成。

5.2.1 文件头

ELF文件头描述了文件格式、平台环境以及文件结构等信息，其数据结构定义为：

```
1  typedef uint16_t Elf32_Half;                                //半字,
2  2字节

3  typedef uint32_t Elf32_Word;                                //字,
4  4字节

5  typedef uint32_t Elf32_Addr;                                //地址,
6  4字节

7  typedef uint32_t Elf32_Off;                                  //偏移,
8  4字节

9  #define EI_NIDENT (16)                                       //魔数长度

10 typedef struct{
11     unsigned char    e_ident[EI_NIDENT];                    //魔数

12     Elf32_Half       e_type;                                  //文件
13     类型

14     Elf32_Half       e_machine;                              //机器类型

15     Elf32_Word       e_version;                              //版本号

16     Elf32_Addr       e_entry;                                //程序入口点

17     Elf32_Off        e_phoff;                                //程序头表文件偏移

18     Elf32_Off        e_shoff;                                //段表文件偏移

19     Elf32_Word       e_flags;                                //平台相关标记
```

```
15      Elf32_Half      e_ehsize;                      //文件头大小

16      Elf32_Half      e_phentsize;                    //程序头表
项大小

17      Elf32_Half      e_phnum;                        //程序头表项个数

18      Elf32_Half      e_shentsize;                    //段表项大
小

19      Elf32_Half      e_shnum;                        //段表项个数

20      Elf32_Half      e_shstrndx;                    //段表字符
串表的段表索引

21 } Elf32_Ehdr;

//文件头
```

第1~4行描述了ELF文件数据结构常用的数据类型，第5行定义了ELF文件头魔数字段的长度，结构体Elf32_Ehdr描述了ELF文件头数据结构。ELF文件头的每个字段的含义如下。

1) e_ident称为ELF文件的魔数，是16字节长的数组，用于标识ELF文件格式、数据存储在字节序、ELF版本等信息。常见的初始值为：

```
71 45 4c 46
01 01 01 00 00 00 00 00 00 00 00 00 00
```

其中前4字节为DEL控制字符和字符“E”、“L”、“F”对应的ASCLL码，对于任意ELF文件，这4字节的值是固定的。第5字节表示文件类

别，0表示无效文件、1表示32位ELF文件、2表示64位ELF文件，我们只使用32位ELF文件，取值ELFCLASS32。第6字节表示字节序，0表示无效格式、1表示小端字节序、2表示大端字节序，我们使用小端字节序，取值ELFDATA2LSB。第7字节表示ELF版本，默认为1，取值EV_CURRENT，表示当前版本号。后面的9字节在ELF标准中未定义，一般用于平台相关的扩展标志。在Linux系统中，第8字节取值ELFOSABI_NONE=0，表示UNIX系统，第9字节取值0，表示系统ABI（Application Binary Interface）版本为0。其他字节默认为0。

2) e_type表示ELF文件类型，0表示无效文件类型、1表示可重定位目标文件、2表示可执行文件、3表示共享目标文件、4表示核心转储文件。我们设计的汇编器输出可重定位目标文件，该字段取值为ET_REL。静态链接器输出可执行文件，该字段取值为ET_EXEC。

3) e_machine表示ELF所在的机器类型，例如3表示Intel 80386体系结构、40表示Arm体系结构。我们生成x86平台的ELF文件，该字段取值为EM_386。

4) e_version表示ELF文件的版本，一般取值1，即EV_CURRENT。

5) e_entry表示ELF文件程序的入口线性地址，一般用于ELF可执行文件。对于可重定位目标文件，该字段设置为0。

6) `e_phoff`表示程序头表在ELF文件内的偏移地址，标识了程序头表在文件内的位置。

7) `e_shoff`表示段表在ELF文件内的偏移地址，标识了段表在文件内的位置。

8) `e_flags`表示ELF文件平台相关的属性，一般默认为0。

9) `e_ehsize`表示ELF文件头的大小，即`sizeof (Elf32_Ehdr) = 52`字节。

10) `e_phentsize`表示程序头表项的大小，即`sizeof (Elf32_Phdr) = 32`字节。

11) `e_phnum`表示程序头表项的个数，因此可以确定程序头表在ELF文件偏移`e_phoff`到`e_phoff + e_phentsize * e_phnum`的数据块中。

12) `e_shentsize`表示段表项的大小，即`sizeof (Elf32_Shdr) = 40`字节。

13) `e_shnum`表示段表项的个数，因此可以确定段表在ELF文件偏移`e_shoff`到`e_shoff + e_shentsize * e_shnum`的数据块中。

14) `e_shstrndx`表示段表字符串表所在段在段表中的索引。这个字段的含义比较复杂，稍后会作详细解释。

使用命令“readelf-h file.o”可以查看ELF文件头的完整信息。

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                   ELF32
  Data:                     2's complement, little
  endian
  Version:                  1 (current)
  OS/ABI:                   UNIX - System V
  ABI Version:              0
  Type:                     REL (Relocatable file)
  Machine:                  Intel 80386
  Version:                  0x1
  Entry point address:      0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 224 (bytes into file)
  Flags:                    0x0
  Size of this header:      52 (bytes)
  Size of program headers:  0 (bytes)
  Number of program headers: 0
  Size of section headers:  40 (bytes)
  Number of section headers: 11
  Section header string table index: 8
```

从输出信息中可以看出，该文件为可重定位目标文件，程序入口点为0。程序头表文件偏移为0，程序头表项大小为0，程序头表项的个数为0，因此不存在程序头表。段表文件偏移为224字节，段表项大小为40字节，段表项个数为11个，因此文件224字节到 $224+11*40=664$ 处保存了段表的内容。

通过ELF文件头，可以访问到ELF内两个最关键的数据结构：段表（Section Header Table）和程序头表（Program Header Table）。

5.2.2 段表

ELF文件内的数据结构，除了段表和程序头表之外，其他都是以段（Section）的方式进行组织的。段表包含了多个段表项，段表项记录了每个段的相关信息，比如段的名称、位置、大小、属性等信息。段表项的数据结构定义为：

```
typedef struct{
    Elf32_Word      sh_name;           // 段名

    Elf32_Word      sh_type;           // 段类型

    Elf32_Word      sh_flags;          // 段标志

    Elf32_Addr      sh_addr;           // 段虚拟基址

    Elf32_Off       sh_offset;         // 段文件偏移

    Elf32_Word      sh_size;           // 段大小

    Elf32_Word      sh_link;           // 段链接信息

1   Elf32_Word      sh_info;           // 段链接信息

2   Elf32_Word      sh_addralign;      // 段对齐方式

    Elf32_Word      sh_entsize;        // 表项大小

} Elf32_Shdr

;                                     // 段表项
```

结构体Elf32_Shdr描述了ELF文件段表项的数据结构，其每个字段的含义如下。

1) `sh_name`是一个4字节偏移量，记录了段名字符串在段表字符串表内的偏移。段表字符串表并非表的形式，而是一个文件块，保存了所有的段表字符串内容，存储在名为“`.shstrtab`”的段中。根据“`.shstrtab`”段的偏移，加上`sh_name`便可以访问到每个段对应的段名字符串，因此欲解析段名必须访问“`.shstrtab`”段。然而，“`.shstrtab`”段的信息也是以段表项存储在段表内的，欲访问“`.shstrtab`”段必须先访问段表。这样，问题好像陷入了一个段表、“`.shstrtab`”段、段表的“死循环”中，而ELF文件头的`e-shstrndx`字段的意义在此便体现出来。`e-shstrndx`字段记录段表字符串表所在的“`.shstrtab`”段对应的段表项在段表内的索引，根据该索引，可以定位到“`.shstrtab`”段表项的位置，计算公式为 $e_shoff + sh_entsize * e_shstrndx$ ，其中`sh_entsize`为段表项的大小。然后根据该段表项取出“`.shstrtab`”段的偏移量`sh_offset`，读出“`.shstrtab`”段的内容，再根据每个段的`sh_name`访问段名字符串的内容。

2) `sh_type`表示段的类型。其中1表示程序段，取值`SHT_PROGBITS`，比如代码段“`.text`”、数据段“`.data`”等。2表示符号表段，取值`SHT_SYMTAB`，如符号表段“`.symtab`”。3表示串表段，取值`SHT_STRTAB`，如段表字符串表段“`.shstrtab`”和串表段“`.strtab`”。8表示

无内容段，取值SHT_NOBITS，比如“.bss”段。9表示重定位表段，取值SHT_REL，比如重定位表段“.rel.text”和“.rel.data”等。

3) sh_flags为段标志，记录段的属性。其中0表示默认属性。1表示段可写，取值SHF_WRITE。2表示段加载后需要为之分配内存空间，取值SHF_ALLOC。4表示段可以执行，取值SHF_EXECINSTR。段标志属性可以进行复合，比如代码段“.text”属性为可分配、可执行、不可写，因此其段属性为SHF_ALLOC|SHF_EXECINSTR。而数据段“.data”或“.bss”段属性为可分配、可写、不可执行，因此其段属性为SHF_ALLOC|SHF_WRITE。对于一般的段，如果需要分配内存则取值SHF_ALLOC，如果不需要分配内存则取默认值0即可。

4) sh_addr表示段加载后的线性地址。在可重定位目标文件内，无法确定段的虚拟地址，故设为默认值0。在可执行文件内，链接器会计算出需要加载的段的线性地址。

5) sh_offset表示段在文件内的偏移，根据此偏移可以确定段的位置，读取段的内容。

6) sh_size表示段的大小，单位为字节。需要注意的是，如果段类型为SHT_NOBITS，段内是没有数据的，那么段大小并非指文件块的大小，而是指段加载后占用内存的大小。

7) `sh_link`和`sh_info`表示段的链接信息，一般用于描述符号表段和重定位表段的链接信息。对于符号表段（段类型为`SHT_SYMTAB`），`sh_link`记录符号表使用的串表所在段（一般是“`.strtab`”）对应段表项在段表内的索引。就像段表项的`sh_name`记录段名字符串在段表字符串表所在段“`.shstrtab`”的偏移一样，符号表项的`st_name`记录符号名字符串在字符串表所在段“`.strtab`”的偏移，而`sh_info`记录符号表内最后一个局部符号的符号表项在符号表内的索引+1，一般恰好是第一个全局符号的符号表项的索引，这可以帮助链接器更快地定位到第一个全局符号。对于重定位表段（段类型为`SHT_REL`），`sh_link`记录重定位表使用的符号表段（一般是“`.symtab`”）对应段表项在段表内的索引，而`sh_info`记录重定位所作用的段对应的段表项在段表内的索引。一般的，重定位表段“`.rel.text`”作用于代码段“`.text`”，“`.rel.data`”作用于数据段“`.data`”。对于其他类型的段，如无特殊要求，`sh_link`和`sh_info`默认为0。

8) `sh_addralign`表示段的对齐方式，对齐规则为 $\text{sh_offset} \% \text{sh_addralign} = 0$ ，即段的文件偏移必须是`sh_addralign`的整数倍。`sh_addralign`取值必须是2的整数幂，如1、2、4、8等，特别地，如果`sh_addralign`取值0或1表示无对齐要求。比如“`.text`”段的文件偏移为118字节，对齐大小为4字节，那么对齐后新的文件偏移为120字节，第118~119字节的两字节数据被“空出”，需要使用数据填充。数据填充一般使用0x00，但是对于代码段数据，使用字节0x90填充会更好，它

们对应汇编指令“nop”，不会影响代码的执行。使用如下公式可以对段偏移进行对齐：

```
offset+=(align-offset%align)%align
```

或

```
offset=offset&align+(offset&(align-1))?align:0
```

我们一般采用第一种方式。在可重定位目标文件中，代码段和数据段的sh_addralign取值一般是4，即按照4字节对齐，其他类型的段sh_addralign取值为1，无对齐要求。在可执行文件中，代码段的sh_addralign取值一般是16，即按照16字节对齐，数据段的sh_addralign取值一般是4，即按照4字节对齐，其他类型的段sh_addralign取值为1，无对齐要求。

9) sh_entsize一般用于保存诸如符号表段、重定位表段时，表示段内保存表的表项大小。例如符号表段“.symtab”内保存的符号表的表项大小为sizeof (Elf32_Sym)=32字节，重定位段“.rel.text”或“.rel.data”内保存的重定位表的表项大小为sizeof

(Elf32_Rel)=8字节。对于其他类型的段，该字段默认值为0，表示段内保存的是非表类型数据。

使用命令“readelf-S file.o”可以查看ELF文件段表的完整信息。

Section Headers:									
	[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk
Inf	Al								
	[0]		NULL	00000000	000000	000000	00		0 0
0		[1] .text	PROGBITS	00000000	000034	00001d	00	AX	0 0
4		[2] .rel.text	REL	00000000	000350	000010	08		9 1
4		[3] .data	PROGBITS	00000000	000054	000000	00	WA	0 0
4		[4] .bss	NOBITS	00000000	000054	000000	00	WA	0 0
4		[5] .rodata	PROGBITS	00000000	000054	00000d	00	A	0 0
1		[6] .comment	PROGBITS	00000000	000061	00002c	01	MS	0 0
1		[7] .note.GNU-stack	PROGBITS	00000000	00008d	000000	00		0 0
1		[8] .shstrtab	STRTAB	00000000	00008d	000051	00		0 0
1		[9] .symtab	SYMTAB	00000000	000298	0000a0	10		10 8
4		[10].strtab	STRTAB	00000000	000338	000015	00		0 0
1									

其中列名Nr表示段表项索引、Name表示段名、Type表示段类型、Addr表示段线性地址、Off表示段文件偏移、Size表示段大小、ES表示段内保存表的表项大小、Flg表示段标志、Lk和Inf表示段链接信息、Al表示段对齐大小。

从输出信息中可以看出，段表的第一项（索引0）保存无效段表项，所有字段初始化为0。代码段“.text”的索引为1，类型为PROGBITS、线性地址为0、文件偏移为0x34、大小为0x1d、段属性为AX，即可分配、可执行（A-Alloc，X-Exec）、按照4字节对齐。数据段“.data”的段属性为WA，即可写、可分配（W-Write，A-Alloc）、按照4字节对齐。“bss”段类型为NOBITS，表示无内容。符号表

段“.symtab”类型为SYMTAB、符号表项大小为0x10、Lk字段为10，对应“.strtab”的段表项，表示符号表使用的串表在该段中、Inf字段为8，表示符号表内第一个全局符号表项的索引（参考后面符号表章节给出的符号表信息）。重定位表段“.rel.text”的类型为REL、重定位表项大小为0x8、Lk字段为9，对应“.symtab”的段表项，表示重定位表使用的符号表在该段中、Inf字段为1，对应“.text”的段表项，表示重定位表作用的段为代码段“.text”。

通过ELF文件的段表，可以访问到ELF所有段的内容和信息，继而访问具体的段。

5.2.3 程序头表

ELF文件的程序头表与段表是相互独立的，它们由ELF文件头统一管理。程序头表管理ELF文件加载后，ELF文件内可加载段到内存镜像的映射关系，一般只有可执行文件包含程序头表。程序头表包含多个程序头表项，程序头表项描述的对象称为“Segment”。为了和段表所描述的段（Section）进行区分，[仅在本节](#)我们约定Segment称为“段”，而Section称为“节”。段描述的是ELF文件加载后的数据块，而节描述的是ELF文件加载前的数据块。一般情况下，段与节之间没有必然的对应关系，但不排除一一对应关系。比如代码节“.text”的加载信息保存在代码段对应的程序头表项中，数据节“.data”的加载信息保存在数据段对应的程序头表项中。有时候，为了简化程序头表项的个数，会把同类型的多个节，甚至整个ELF文件作为一个段加载，这样段与节之间就没有对应关系了。

程序头表描述的加载信息之所以可以如此灵活，与程序头表项的信息是分不开的。程序头表项记录了每个段的相关信息，比如段的类型、对应文件的位置、大小、属性等信息，这些信息与段表描述的节信息是相互独立的。程序头表项的数据结构定义为：

```
1 typedef struct{
2     Elf32_Word      p_type;           //段类型
```



```

3      Elf32_Off      p_offset;           //段文件偏移
4      Elf32_Addr     p_vaddr;           //段虚拟地址
5      Elf32_Addr     p_paddr;           //段物理地址
6      Elf32_Word     p_filesz;          //段在文件中的大小
7      Elf32_Word     p_memsz;           //段需要的内存大小
8      Elf32_Word     p_flags;           //段标志
9      Elf32_Word     p_align;           //段对齐方式

10 } Elf32_Phdr;

                                     //程序头表项

```

结构体Elf32_Phdr描述了ELF文件程序头表项的数据结构，其每个字段的含义如下。

1) p_type表示段的类型，这里我们只关心可加载的段，取值为PT_LOAD。

2) p_offset表示段对应的内容在文件内的偏移。

3) p_vaddr表示段在内存的线性地址。

4) p_paddr表示段在内存的物理地址，由于现代操作系统中使用了分页机制，因此不需要关心段的物理地址，一般该字段值与p_vaddr相同。但是对于未使用分页机制的系统，该字段可以另行设置。

5) `p_filesz`表示段在文件内的大小。

6) `p_memsz`表示段在内存的大小。我们可以总结出字段2、3、5、6表达的含义为：ELF文件内从`p_offset`开始的`p_filesz`大小的数据块被加载到内存以`p_vaddr`开始的`p_memsz`大小的内存块中。由于段的内容需要完整加载到内存，因此`p_memsz`字段的值一般等于`p_filesz`。但是对于类型为`SHT_NOBITS`的节，在ELF文件内不存在数据。例如“.bss”节，它在ELF文件内的大小为0，如果加载到内存后大小大于0，那么其对应的程序头表项的`p_filesz`等于0，而`p_memsz`为一个正整数。

7) `p_flags`表示段标志，与段表项的`sh_flags`类似，描述了段的属性（权限）。其中1表示可执行，取值为`PF_X`。2表示可写，取值为`PF_W`。4表示可读，取值为`PF_R`。段标志可以进行复合，例如代码节“.text”对应的程序头表项的段标志为`PF_R|PF_X`，即可读可执行。数据节“.data”对应的程序头表项的段标志为`PF_R|PF_W`，即可读可写。

8) `p_align`表示段对齐方式，对齐规则为`p_vaddr%p_align=0`，即段的线性地址必须是`p_align`的整数倍。一般情况下，`p_align`取值为`0x1000=4096`，即Linux操作系统默认的页大小。

如表5-17所示，文件内有两个段需要加载，假设默认加载的线性地址为`0x08048000`。第一个段Seg1的文件偏移为`0x34`，加载后大小为

0x1030。由于是第一个段，因此加载地址为0x08048000。第二个段Seg2的文件偏移为0x1064，加载后大小为0x64。由于Seg1加载后占用了0x0804800~0x08049030的地址空间，将0x08049030按照0x1000对齐后得到Seg2的加载地址为0x0804a000。我们发现按照这样的加载方式，共需要占用3个物理页框，对应线性地址空间范围为0x08048000~0x0804a000。

表5-17 段地址对齐

段	p_paddr	p_offset	p_memsz
Seg1	0x08048000	0x34	0x1030
Seg2	0x0804a000	0x1064	0x64

Linux系统中，当一个段大小不是页的整数倍时，将该段的尾部与下一个段的开始部分放在同一个物理页框内，以减少物理内存的消耗。由于Seg1与Seg2对应的内存块大小总和为0x1094，即使加上Seg1之前的文件大小0x34，总大小为0x10c8也不超过两个页框大小，因此使用两个物理页框足以完成段的加载。其基本思想是，将ELF文件从偏移0处开始到最后一个需要加载的段结束位置的地址空间，按照页大小进行逻辑划分，形成多个页，每个页都被加载到物理页框中，然后将每个物理页框映射到线性地址空间的页去。如果物理页框内保存了N个段的内容，那么需要向线性地址空间的页映射N次。

如图5-5所示，描述了ELF文件内需要加载的内容对应的段Seg1和Seg2的布局（如果加载的内容包含“.bss”节，则以其加载后的大小

p_memsz为准)。将ELF文件按照页大小划分，需要加载的段可以保存在两个逻辑页内，其中Seg1段被划分到两个逻辑页中。每个逻辑页都被独立加载到物理页框内，这样逻辑块1对应的物理页框只保存了Seg1的上半部分内容Seg1-1，而逻辑块2对应的物理页框保存了Seg1的下半部分内容Seg1-2和Seg2的全部内容。通过页映射将物理页框再映射到虚拟内存页面，逻辑块1对应的物理页框映射到线性地址空间0x08048000~0x08049000，而逻辑块2包含两个段的内容，因此需要映射两次，分别映射到线性地址空间0x08049000~0x0804a000和0x0804a000~0x0804b000。我们可以看出在虚拟内存中Seg1占据的线性地址空间为0x08048034~0x08049064，Seg2占据的线性地址空间为0x0804a064~0x0804a0c8。当然，我们不可否认，由于逻辑块2对应的物理块的多次映射，在0x08049064处保存了Seg2的内容，同理在0x0804a000处保存了Seg1-2的内容。但是由于段按照页大小对齐的原则，要求每个虚拟内存页仅保存同一个段的内容，对于由于多次映射导致页内数据“重复”的问题并不关心。

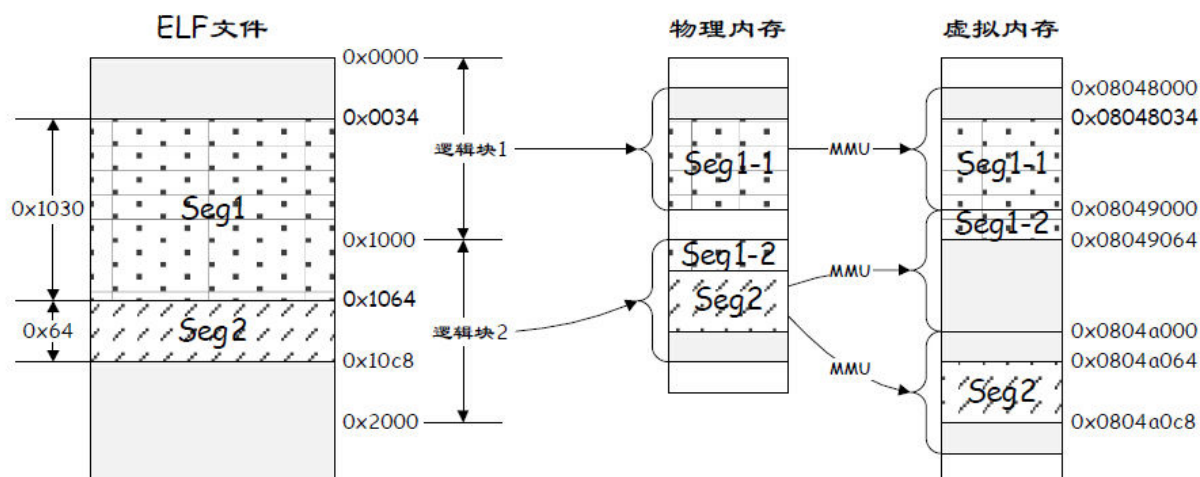


图5-5 改进的段对齐方式

通过上述段地址的对齐方式，原本需要使用3个物理页框保存的段，只需要两个物理页框保存即可，减少了物理内存的消耗，而在线性地址空间内仍需要3个虚拟内存页的大小。但是这样的段地址对齐方式违反了 $p_vaddr \% p_align = 0$ 的原则，因此改进的段地址对齐规则描述为 $p_vaddr \% p_align = p_offset \% p_align$ ，即段的线性地址与段对应文件内容偏移相对于段对齐方式取模同余。使用如下公式对 p_vaddr 进行对齐。

$$p_vaddr += (p_align - p_vaddr \% p_align) \% p_align + p_offset \% p_align$$

即将 p_vaddr 按照 p_align 正常对齐后，然后累加上 p_offset 对 p_align 的模。例如Seg1的初始加载地址 $p_vaddr = 0x08048000$ ，根据 $p_align = 0x1000$ 对齐后仍为 $0x08048000$ ，累加 $p_offset \% p_align = 0x34 \% 0x1000 = 0x34$ 后得到对齐后的 $p_vaddr = 0x08048034$ 。同理，Seg2的初始加载地址为Seg1加载后的下一个字节地址， $p_vaddr = 0x08048034 + 0x1030 = 0x08049064$ ，根据 $p_align = 0x1000$ 对齐后为 $0x0804a000$ ，累加 $p_offset \% p_align = 0x1064 \% 0x1000 = 0x64$ 得到对齐后的 $p_vaddr = 0x0804a064$ 。

使用命令“`readelf -l file`”可以查看ELF文件程序头表的完整信息。

Program Headers:						
Flg	Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz
	Align					
R E	LOAD	0x000000	0x08048000	0x08048000	0x84fd2	0x84fd2
	0x1000					
RW	LOAD	0x085f8c	0x080cdf8c	0x080cdf8c	0x007d4	0x02388
	0x1000					
R	NOTE	0x0000f4	0x080480f4	0x080480f4	0x00044	0x00044
	0x4					
R	TLS	0x085f8c	0x080cdf8c	0x080cdf8c	0x00010	0x00028
	0x43					
RW	GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000
	0x4					
R	GNU_RELRO	0x085f8c	0x080cdf8c	0x080cdf8c	0x00074	0x00074
	0x1					

其中列名**Type**表示程序头表项类型、**Offset**保存段数据块相对文件开始处的偏移、**VirtAddr**表示线性地址、**PhysAddr**表示物理地址、**FileSiz**表示段内容在文件内的大小、**MemSiz**表示段加载后的内存大小、**Flg**表示段标志、**Align**表示段对齐大小。

从输出信息中可以看出，程序头表包含两个可加载（LOAD）类型的段。第一个段是从文件内偏移为0处开始，大小为0x84fd2的文件块，段标志为RE，即可读可执行，由此可以推断该段包含了代码节“.text”，并且将文件头的内容也一起加载了。加载后的虚拟地址为0x08048000，内存大小仍为0x84fd2，对齐方式为0x1000。第二个段开始于文件偏移0x085f8c处，是大小为0x007d4的文件块，段标志为RW，即可读可写。段加载后的虚拟地址为0x080cdf8c，内存大小为0x02388>0x007d4，由此可以推断该段包含了“.bss”节，导致加载后的内存大小大于文件块大小。

通过ELF文件的程序头表，可以为加载器提供可执行文件的详细加载信息，包括哪些文件内容需要加载、加载到进程地址空间的哪个位置、页面的权限等信息。

5.2.4 符号表

ELF文件的符号表保存了程序中的符号信息，包括程序中的文件名、函数名、全局变量名等。符号表一般保存在名为“.strtab”的段内，该段对应段表项的类型为SHT_SYMTAB。符号表包含多个符号表项，每个符号表项记录了符号的名称、位置、类型等信息。符号表项的数据结构定义为：

```
1  typedef struct{
2      Elf32_Word      st_name;                //符号名

3      Elf32_Addr      st_value;              //符号值

4      Elf32_Word      st_size;               //符号大小

5      unsigned char    st_info;              //符号类型

6      unsigned char    st_other;            //无用字段

7      Elf32_Section    st_shndx;            //符号所在段

8  } Elf32_Sym;

                                     //符号表项
```

结构体Elf32_Sym描述了ELF文件符号表项的数据结构，其每个字段的含义如下。

1) `st_name`是一个4字节字段，记录了符号名字字符串在字符串表的偏移。与段表字符串表类似，字符串表也不是表的形式，而是一个文件块，保存了所有的符号名字字符串内容，存储在名为“`.strtab`”的段中。根据“`.strtab`”段的偏移，加上`st_name`便可以访问每个符号对应的符号名字字符串，因此欲解析符号名必须访问“`.strtab`”段。与段表中提到的“`.shstrtab`”段一起，我们在串表一节会对它们作详细描述。

2) `st_value`记录了符号的值，一般在可重定位目标文件内，该值记录了符号相对于所在段基址的偏移量，而在可执行文件内，该值记录了符号的线性地址。

3) `st_size`表示符号的大小，单位为字节。

4) `st_info`大小为一个字节，表示符号类型相关的信息，其低4位表示符号的类型使用宏`ELF32_ST_TYPE`获取，高4位表示符号的绑定信息，使用宏`ELF32_ST_BIND`获取。符号类型为0时表示未知类型，取值`STT_NOTYPE`。1表示数据对象，比如变量、数组等，取值`STT_OBJECT`。2表示函数，取值`STT_FUNC`。3表示段，取值`STT_SECTION`。4表示文件名，取值`STT_FILE`。符号绑定信息为0时表示局部符号，取值`STB_LOCAL`。1表示全局符号，取值`STB_GLOBAL`。2表示弱符号，取值`STB_WEAK`，为了简化问题的讨论，我们不关心弱符号相关的ELF文件信息。

5) `st_other`没有实际含义，默认为0。

6) `st_shndx`表示符号所在段对应的段表项在段表内的索引，一般取值为正整数。当该字段为0时，表示符号未定义，取值 `SHN_UNDEF`。该字段为0xfff1时，表示符号为绝对值，比如文件名，取值 `SHN_ABS`。该字段为0xfff2时，表示符号在COMMON块内，取值 `SHN_COMMON`，特别的，此时符号的`st_value`表示符号的对齐属性。COMMON块与“弱符号”的概念相关，我们不作深入讨论，感兴趣的读者可以检索弱符号的资料深入了解。

使用命令“`readelf-s file.o`”可以查看ELF文件符号表的完整信息。

Symbol table '.symtab' contains 10 entries:							
Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	FILE	LOCAL	DEFAULT	ABS	hello.c
2:	00000000	0	SECTION	LOCAL	DEFAULT	1	
3:	00000000	0	SECTION	LOCAL	DEFAULT	3	
4:	00000000	0	SECTION	LOCAL	DEFAULT	4	
5:	00000000	0	SECTION	LOCAL	DEFAULT	5	
6:	00000000	0	SECTION	LOCAL	DEFAULT	7	
7:	00000000	0	SECTION	LOCAL	DEFAULT	6	
8:	00000000	29	FUNC	GLOBAL	DEFAULT	1	main
9:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf

其中列名Num表示符号表项索引、Value表示符号值、Size表示符号大小、Type表示符号类型、Bind表示符号绑定信息、Vis信息不必关心、Ndx表示符号所在段、Name为符号名。

从输出信息中可以看出，符号表的第一项（索引0）保存无效符号表项，所有字段初始化为0。索引为1的符号表项表示文件

名“hello.c”，其类型为TYPE，绑定信息为LOCAL，st_shndx值为SHN_ABS。索引为8的符号表项表示函数名“main”，类型为FUNC，大小为29Byte，绑定信息为GLOBAL，st_shndx=1表示符号在代码段“.text”内（见5.2.2节ELF文件的段表信息）。索引为9的符号表项表示函数名“printf”，st_shndx=SHN_UNDEF，表示符号未定义（外部符号），因此符号类型为NOTYPE，即未知类型，外部符号绑定信息一般都是GLOBAL。

通过ELF文件的符号表，可以访问所有符号的信息，其中符号名、符号值和绑定信息对链接器至关重要。

5.2.5 重定位表

重定位表常见于可重定位目标文件内，对于静态链接生成的可执行文件，一般不包含重定位表，动态链接生成的可执行文件不在我们讨论的范围内。重定位表一般保存在以名为“.rel”开头的段内，该段对应段表项的类型为SHT_REL。ELF文件需要重定位的段，一般都对应一个重定位表。比如代码段“.text”的重定位表保存在“.rel.text”段内，数据段“.data”的重定位表保存在“.rel.data”内。重定位表包含多个重定位表项，每个重定位表项记录一条重定位信息，包括重定位的符号、位置、类型等信息。重定位表项的数据结构定义为：

```
1  typedef struct{
2      Elf32_Addr      r_offset;           // 重定位地址

3      Elf32_Word      r_info;           // 重定位类型和符号

4  } Elf32_Rel;

                                     // 重定位表项
```

结构体Elf32_Rel描述了ELF文件重定位表项的数据结构，其每个字段的含义如下。

1) r_offset表示重定位地址，对于可重定位目标文件来说，表示重定位位置相对于被重定位段的基址的偏移。而对于可执行文件或共

享目标文件来说，表示重定位位置对应的线性地址，这与动态链接相关，我们不作深入讨论。

2) `r_info`描述了重定位类型和符号，其低8位表示重定位类型，使用宏`ELF32_R_TYPE`获取，高24位表示重定位符号对应的符号表项在符号表内的索引，使用宏`ELF32_R_SYM`获取。不同的处理器体系结构都有属于自己的一套重定位类型，对于x86体系结构而言，静态链接中常见的重定位类型有两种：绝对地址重定位（取值`R_386_32`）和相对地址重定位`R_386_PC32`。每条重定位信息的含义为使用`r_info`记录的符号的线性地址，根据重定位类型更新`r_offset`处的内存信息。关于不同重定位类型的实现细节，在第7章中会进行详细描述。

使用命令“`readelf-r file.o`”可以查看ELF文件重定位表的完整信息。

Relocation section '.rel.text' at offset 0x350 contains 2 entries:				
Offset	Info	Type	Sym.Value	
Sym.Name				
0000000a	00000501	R_386_32	00000000	
.rodata				
00000012	00000902	R_386_PC32	00000000	printf

其中列名`Offset`表示重定位地址、`Info`表示重定位信息、`Type`表示重定位类型、`Sym.Value`表示重定位符号的值`st_value`、`Sym.Name`表示重定位符号的名称。

从输出信息中可以看出，重定位表的第二项描述了使用符号`printf`对代码段“`.text`”的0x12处的内存进行重定位，重定位类型为相对地址

重定位。这是因为在可重定位目标文件内，`printf`是外部符号，无法确定它的线性地址，因此对`printf`的`call`指令的操作数无法确定，必须由链接器根据该重定位信息重新计算`call`指令的操作数。

根据ELF文件的重定位表描述的重定位信息，链接器对目标文件内的数据和代码内容进行修正，保证了可执行文件内的代码和数据的完整性。

5.2.6 串表

ELF文件内的段表和符号表需要记录段名和符号名，这些名称都是字符串。然而，段表项和符号表项都是固定长度的数据结构，无法存储不定长的字符串。因此ELF文件将名称字符串内容集中存放在一个段内，称为串表。这样段表项或符号表项只需要记录段名字符串或符号名字符串在对应串表内的位置即可。虽然存储的字符串内容称为串表，但并非“表”的形式，而是一块文件区域。

使用命令“`hexdump-C file.o`”可以查看ELF文件的所有信息。

00000000	7f 45 4c 46 01 01 01 00	00 00 00 00 00 00 00 00	.ELF.....
00000010	01 00 03 00 01 00 00 00	00 00 00 00 00 00 00 00
00000020	e0 00 00 00 00 00 00 00	34 00 00 00 00 00 28 004....(
00000030	0b 00 08 00 55 89 e5 83	e4 f0 83 ec 10 b8 00 00U.....
00000040	00 00 89 04 24 e8 fc ff	ff ff b8 00 00 00 00 c9\$......
00000050	c3 00 00 00 48 65 6c 6c	6f 20 57 6f 72 6c 64 21Hello World!
00000060	00 00 47 43 43 3a 20 28	55 62 75 6e 74 75 2f 4c	..GCC: (Ubuntu/L
00000070	69 6e 61 72 6f 20 34 2e	34 2e 34 2d 31 34 75 62	inaro 4.4.4-14ub
00000080	75 6e 74 75 35 29 20 34	2e 34 2e 35 00 00 2e 73	untu5) 4.4.5...s
00000090	79 6d 74 61 62 00 2e 73	74 72 74 61 62 00 2e 73	ymtab..strtab..s
000000a0	68 73 74 72 74 61 62 00	2e 72 65 6c 2e 74 65 78	hstrtab..rel.tex
000000b0	74 00 2e 64 61 74 61 00	2e 62 73 73 00 2e 72 6f	t..data..bss..ro
000000c0	64 61 74 61 00 2e 63 6f	6d 6d 65 6e 74 00 2e 6e	data..comment..n
000000d0	6f 74 65 2e 47 4e 55 2d	73 74 61 63 6b 00 00 00	ote.GNU-stack...
000000e0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
*			
00000330	00 00 00 00 10 00 00 00	00 68 65 6c 6c 6f 2e 63hello.c
00000340	00 6d 61 69 6e 00 70 72	69 6e 74 66 00 00 00 00	.main.printf....
00000350	0a 00 00 00 01 05 00 00	12 00 00 00 02 09 00 00
00000360			

在前面给出的段表信息中，“.shstrtab”段的文件偏移为0x8d，大小为0x51。段内第一个字节为0x00，表示空串“”。后面依次为段名字符串“.symtab”“.strtab”“.shstrtab”“.rel.text”“.data”“.bss”“.rodata”“.comment”“

.note.GNU-stack”。因此对于“.rel.text”段，其段表项的字段sh_name=0xa8-0x8d=0x1b。而对于“.text”段，其段名是“.rel.text”的后缀，因此可以共享字符串存储，对应的段表项的字段sh_name=0xac-0x8d=0x1f。

而“.strtab”段的文件偏移为0x338，大小为0x15。段内第一个字节为0x00，表示空串“”。后面依次为符号名字符串“hello.c”“main”“printf”。因此对于符号printf，其符号表项的字段st_name=0x346-0x338=0x0e。

ELF文件将字符串内容保存到串表段内，这样使用字符串的文件结构只需要记录字符串在哪个段的哪个位置即可。

综上所述，我们可以总结出常见ELF文件结构之间的关系。

如图5-6所示，通过ELF文件头的e_ph*字段可以定位到程序头表。同样地，通过e_sh*可以定位到段表。段表内记录了串表、符号表、重定位表对应的段，以及代码段和数据段的信息，欲取得这些段的段名，必须通过ELF文件头结构提供的e_shstrndx字段找到“.shstrtab”的段表项，继而定位到该段的数据，取得其保存的所有段的名称sh_name。通过符号表段“.symtab”的段表项内的sh_link字段记录的“.strtab”段在段表的索引找到字符串表，继而取得该表内保存的所有符号的名称st_name。重定位表段（“.rel.data”和“.rel.text”）通过其段表项内的sh_info字段找到被重定位的段，并通过重定位项内的r_offset找到需要

重定位的位置。另外，重定位表还通过其段表内的sh_link字段找到被重定位符号所在的符号表段，并结合重定位项的r_info字段保存的被重定位符号在符号表内的位置找到被重定位的符号信息。

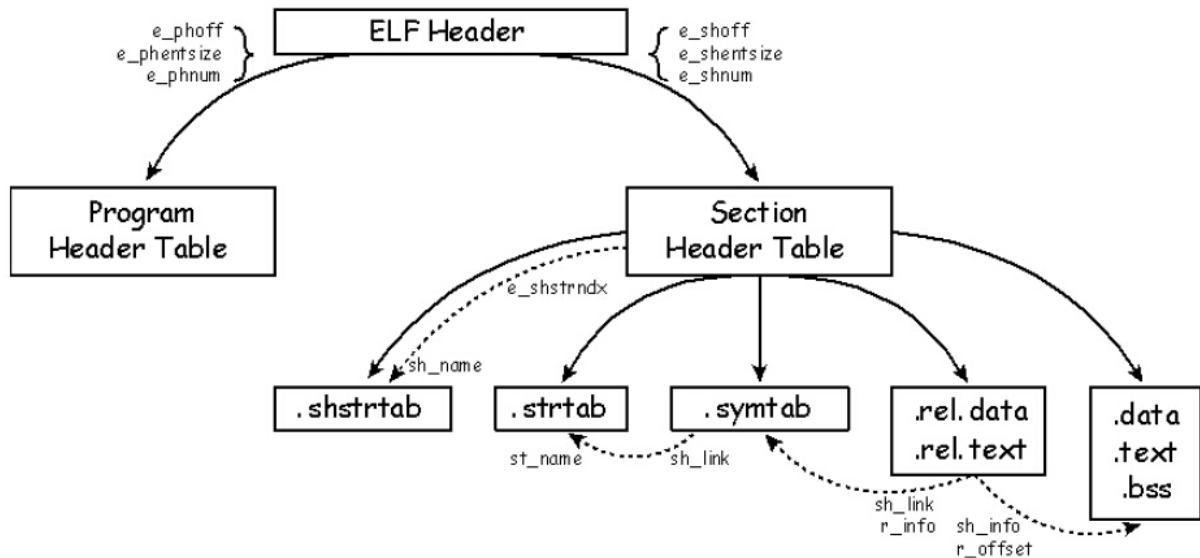


图5-6 ELF文件结构关联

5.3 本章小结

本章详细讨论了x86指令格式和ELF文件格式的相关知识。通过对x86指令结构的解析，了解了指令中的指令前缀、操作码、ModR/M字段、SIB字段、偏移、立即数的功能和含义，并对比了AT&T汇编格式与Intel汇编格式的不同，以帮助理解Linux下的x86汇编语言。通过对ELF文件结构的解析，了解了文件头、段表、程序头表、符号表、重定位表、串表的含义和功能，彻底理清了可重定位目标文件和可执行文件的内容和细节。

在接下来的汇编器构造中，大多数工作都是集中在分析和收集汇编语言中与可重定位目标文件结构相关的信息上，而分析汇编指令结构，以及将之翻译为二进制代码与前面描述的x86指令格式息息相关。在最后的链接器构造中，更需要根据ELF文件格式解析可重定位目标文件的内容，生成可执行文件。链接过程中的重定位操作也需要用到ELF文件的重定位表结构，以及x86指令结构的相关知识。

第6章 汇编器构造

不识庐山真面目，只缘身在此山中。

——《题西林壁》

从字面上来看，汇编器和编译器好像是两种完全不同的事物，实际上它们之间有着很大的相似性。与其称为汇编器，倒不如称作“汇编语言编译器”更为合适。编译器将高级语言翻译为汇编语言，而汇编器将汇编语言翻译为二进制语言。结合前面介绍的现代编译器的结构——前端、优化器和后端，汇编器和编译器拥有相似的前端结构，即它们的词法分析器和语法分析器结构基本相同，差别在于输入的数据形式不同。汇编器的词法分析器的输入是汇编语言的词法记号，语法分析器的输入是汇编语言文法。

正如构造编译器时需要清晰了解自定义语言的特性那样，构造汇编器时也要清晰了解待处理的汇编语言特性。我们的目的并不是构造一个完善的工业化汇编器，拥有处理所有形式汇编指令的能力。实际上只需要处理已实现的编译器生成的汇编代码所涉及的指令，便达到了学习构造汇编器的目的。

结合编译器生成的汇编代码，我们对自定义的汇编语言的特性描述如下：

1) 符号声明。支持NASM格式的数据定义、**section**段声明、**global**全局符号声明、**equ**宏声明等。汇编语言的标识符包含编译器生成的临时符号（以“@L”加数字构成）以及段名（以‘.’开始的字符串），因此汇编语言标识符允许出现特殊符号‘@’和‘.’。

2) 常量。支持整数和字符串常量，其中整数仅限十进制整数，而字符串常量不包含转义字符，这是因为编译器生成数据段时将整数统一按照十进制输出，将字符串转化为仅包含可见字符的NASM格式的字符串。

3) 寻址模式。支持立即寻址、寄存器寻址、寄存器间址、间接寻址、基址+偏移寻址、基址+变址寻址的寻址方式。我们的编译器输出的汇编指令中不存在基址+变址+偏移的寻址方式，在此不作考虑。

4) 指令。支持编译器输出的所有指令。其中包含双操作指令 **mov**、**cmp**、**add**、**sub**、**and**、**or**、**lea**，单操作数指令 **call**、**int**、**imul**、**idiv**、**neg**、**inc**、**dec**、**jmp**、**je**、**jne**、**sete**、**setne**、**setg**、**setge**、**setl**、**setle**、**push**、**pop**，无操作数指令 **ret**。

5) 其他。支持分号开始的单行注释。

参考图2-12描述的汇编器的结构设计，我们接下来详细阐述汇编器每个功能模块的实现。

6.1 词法分析

与编译器词法分析的过程相同（参考图3-1），汇编器的词法分析也需要扫描器顺序读取汇编语言源文件的字符，然后与汇编语言词法记号的有限自动机匹配，得到汇编语言的词法记号。

因此，在汇编器的词法分析阶段，我们只需要弄清所需的词法记号，以及识别词法记号的有限自动机，便可以依样画葫芦地构造汇编器的词法分析器。

6.1.1 词法记号

结合前面对自定义汇编语言特性的描述，我们设计的汇编语言词法记号如下：

1) 符号声明。支持NASM格式的数据定义、section段声明、global全局符号声明、equ宏声明等。NASM格式的数据定义中，使用db、dw、dd描述单位内存的大小，使用times表示数据内容的重复次数，以及使用逗号分隔不同的数据内容（如数字和字符串），因此涉及的词法记号有关键字db、dw、dd、times和分隔符‘，’。另外，段声明、全局符号声明和宏声明涉及了关键字section、global和equ。最后，汇编语言中经常使用标签符号（标识符后紧跟符号‘：’），因此‘：’也是词法记号。

2) 常量。支持整数和字符串常量，涉及的词法记号有数字常量num和字符串常量str。其中数字常量支持十进制整数，因此允许出现正负号‘+’和‘-’。与常量对应的变量使用标识符表示，因此标识符id也是词法记号，只不过汇编语言的标识符允许字符‘@’和‘.’出现。

3) 寻址模式。支持立即寻址、寄存器寻址、寄存器间址、间接寻址、基址+偏移寻址、基址+变址寻址的寻址方式。在各种寻址模式中，经常会用到寄存器进行寻址，因此所有的寄存器名都是关键字。

由于我们的编译器生成的汇编指令中只使用了8位和32位寄存器，为了简化，我们定义了如下寄存器名关键字：8位寄存器al、cl、dl、bl、ah、ch、dh、bh，以及32位寄存器eax、ecx、edx、ebx、esp、ebp、esi、edi。

4) 指令。支持编译器输出的所有指令。指令助记符不能作为一般的标识符使用，因此全部是关键字。包括双操作指令mov、cmp、add、sub、and、or、lea，单操作数指令call、int、imul、idiv、neg、inc、dec、jmp、je、jne、sete、setne、setg、setge、setl、setle、push、pop，无操作数指令ret。

5) 其他。支持分号开始的单行注释。为了方便处理，我们假定编译器生成的汇编语言是没有错误的。即便如此我们也需要词法记号err表示无效的词法记号（如注释），词法分析器或语法分析器都自动忽略这个词法记号。同样地，还有一个词法记号end，它表示文件结束。

于是，我们得到自定义汇编语言所有的词法记号，如表6-1所示。

表6-1 汇编词法记号

类别	词法记号	含义	类别	词法记号	含义
特殊	err	错误记号	关键字	i_sub	sub
	end	文件结束		i_add	add
标识符	id	标识符		i_and	and
常量	num	数字		i_or	or
	str	字符串		i_lea	lea
关键字	kw_sec	section		i_call	call
	kw_glb	global		i_int	int
	kw_equ	equ		i_imul	imul
	kw_times	times		i_idiv	idiv
	kw_db	db		i_neg	neg
	kw_dw	dw		i_inc	inc
	kw_dd	dd		i_dec	dec
	br_al	al		i_jump	jmp
	br_cl	cl		i_je	je
	br_dl	dl		i_jne	jne
	br_bl	bl		i_sete	sete
	br_ah	ah		i_setne	setne
	br_ch	ch		i_setg	setg
	br_dh	dh		i_setge	setge
	br_bh	bh		i_setl	setl
	dr_eax	eax		i_setle	setle
	dr_ecx	ecx		i_push	push
	dr_edx	edx		i_pop	pop
	dr_ebx	ebx		i_ret	ret
	dr_esp	esp	界符	add	+
	dr_ebp	ebp		sub	-
	dr_esi	esi		comma	,
	dr_edi	edi		lbrac	[
	i_mov	mov		rbrac]
	i_cmp	cmp			

同样地，我们使用一个枚举类型记录所有的词法记号标签，为后面的代码提供符号定义。

1

/*

2

词法记号标签

```

3  */
4  enum Tag

5  {
        ERR,                                //错误, 异常

6        END,                                //文件结束标记

7        ID,                                //标识符

8        NUM, STR,                           //常量

9        BR_AL, BR_CL, BR_DL, BR_BL,         //8位寄存器

10       BR_AH, BR_CH, BR_DH, BR_BH,
11       DR_EAX, DR_ECX, DR_EDX, DR_EBX,      //32位寄存器

12       DR_ESP, DR_EBP, DR_ESI, DR_EDI,
13       I_MOV, I_CMP, I_SUB, I_ADD, I_AND, I_OR, I_LEA, //双操作数指令

14       I_CALL, I_INT, I_IMUL, I_IDIV, I_NEG, I_INC, I_DEC, //单操作数指令

15       I_JMP, I_JE, I_JNE,
16       I_SETE, I_SETNE, I_SETG, I_SETGE, I_SETL, I_SETLE,
17       I_PUSH, I_POP,
18       I_RET,
19       KW_SEC, KW_GLB, KW_EQU, KW_TIMES,    //声明

20       KW_DB, KW_DW, KW_DD,
21       ADD, SUB, COMMA, LBRAC, RBRAC, COLON  //界符

22 };

```

确定了词法分析器的词法记号后，接下来定义有限自动机对词法记号进行解析。

6.1.2 有限自动机

汇编器的词法记号相对于编译器要简单很多，因此汇编语言词法记号的有限自动机形式也相对简单。我们重点介绍汇编器中与编译器差别较大的有限自动机的构造。

1.标识符

由于编译器生成的汇编代码中，字符‘@’用于修饰自动生成的符号，字符‘.’用于引导段名，因此汇编器的标识符中允许出现这两个字符。因此，汇编语言标识符有限自动机如图6-1所示。

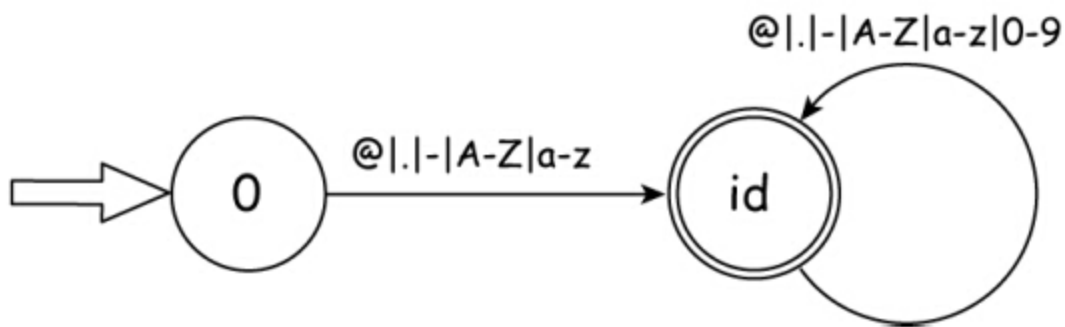


图6-1 标识符有限自动机

2.关键字

汇编语言包含大量的关键字，这是因为除了一般的关键字外，汇编指令的助记符、寄存器名也必须是唯一的。除了在数量上汇编器比

编译器的关键字多之外，对于关键字的识别方式汇编器和编译器完全相同。这里读者可以参考编译器中关键字章节描述的内容。

3.常量

汇编器涉及的常量词法记号有两种：数字常量和字符串常量，且形式比编译器的更加简单。

对于数字常量，我们设计的词法分析器仅考虑十进制非负整数，至于正负号交给语法分析器处理。因此数字词法记号的定义为数字字符0~9的任意组合。其有限自动机结构如图6-2所示。

对于字符串常量，由于汇编语言使用了数字代替了字符串内出现的特殊字符（换行、表符等），因此汇编语言字符串有限自动机内不需要考虑转义字符的情况。其有限自动机结构如图6-3所示。

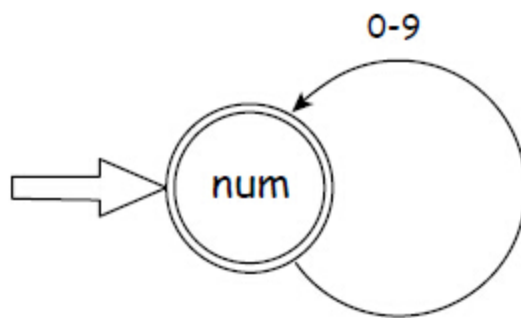


图6-2 整数有限自动机

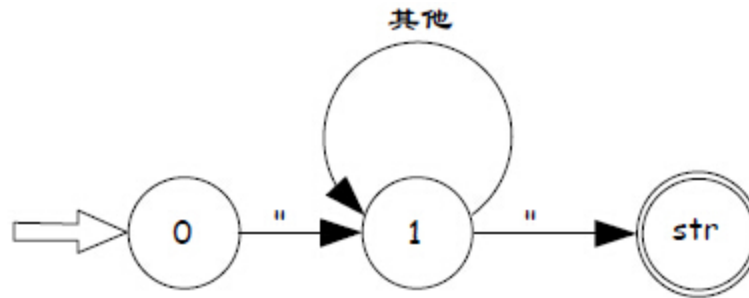


图6-3 字符串有限自动机

4.界符

在我们定义的汇编语言中，只有单字节界符存在，且只有6个，分别是表示整数符号的‘+’和‘-’、分隔符‘，’、指令的内存操作数所需的符号‘[’和‘]’、标签使用的‘：’。其识别方式与编译器的界符相同。

5.无效词法记号

我们定义的汇编语言涉及的无效词法记号也包含空白字符和注释，其中空白字符的识别方式与编译器完全相同，而注释“退化”为以‘；’开始的单行注释。其有限自动机结构如图6-4所示。

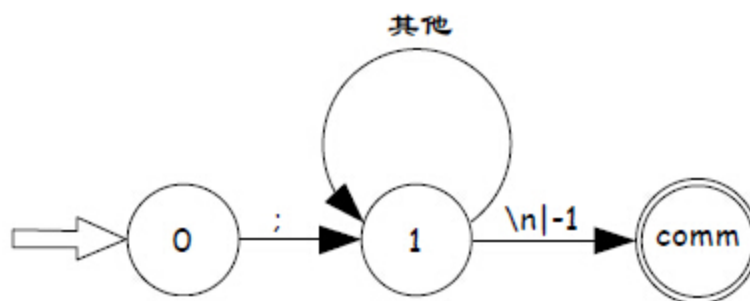


图6-4 注释有限自动机

根据以上对汇编语言词法记号有限自动机的描述，结合编译器章节对词法分析器、解析器的构造原理，不难构造出汇编器的词法分析器。

6.2 语法分析

我们的汇编器语法分析器也是使用递归下降子程序的方式实现的，与编译器章节描述的语法分析类似，明确语言的文法定义是构造语法分析器的前提。

6.2.1 汇编语言程序

相比于高级语言，汇编语言在语法结构上要简单许多。宏观上来看，汇编语言程序包含两个重要的组成部分：声明和指令。

例如汇编程序：

```
1  section .data                                //数据段声明
2  array times 256 db 0                        //数组数据定义
3  x db 100                                    //变量数据定义
4  section .text                                //代码段声明
5  global main                                //全局符号声明
6  main:                                       //标签数据定义
7  ret                                         //指令
```

整个汇编语言程序是声明和指令的“无限”组合，因此使用如下产生式可以描述汇编语言程序。

```
<program>-><segment><program> | ε
<segment>-><dec> | <inst>
```

声明部分包含：

- 1) 段声明，用于声明一个新的段，例如`section.text`。
- 2) 全局符号声明，用于声明全局符号，例如`global main`。
- 3) 数据定义，用于定义数据或标签等。形如`array times 256 db 0`等。

数据定义的形式可能有很多种，不过它们都是以标识符ID开始，不同于段声明以KW_SEC开始，以及全局符号声明以KW_GLB开始。

指令部分则是以不同的汇编指令助记符开始的，可以与声明完全区分开。因此我们可以使用如下的产生式序列表达汇编语言程序。

```
<program>->KW_SEC ID <program>  
  
      | KW_GLB ID <program>  
  
      | ID <lbtail> <program>  
  
      | <inst> <program>  
  
      |  $\epsilon$ 
```

其中非终结符<lbtail>表示以标识符开始的数据定义的后半部分。由于以上产生式不包含公共的左公因子，因此可以同时出现在<program>右侧的产生式中。

按照编译器章节对递归下降子程序构造方式的描述，<program>对应的子程序的代码为：

```
1 void Parser::program
2 {
3     switch(look->tag){
4         case END
5             :
6                 //文件结束，停止语法分析
7                 return;
8                 case KW_SEC
9                 :
10                    //段声明
11                    match(ID
12                    );
13                    break;
14                    case KW_GLB
15                    :
16                    //全局符号声明
17                    match(ID
18                    );
19                    break;
20                    caseID
21                    :
22                    //数据定义
23                    lbtail
24                    ();
25                    break;
26                    default:
27                        //指令
28                        inst
29                        ();
30                    }
31    }
32    program
33    ();
34 }
```

6.2.2 数据定义

鉴于汇编语言中数据定义形式的多样性，我们将其公共首部ID提取出来，剩余的尾部统称为<lbtail>。汇编语言的数据定义有以下几种形式：

- 1) 纯标签，用于表示一个地址，如“main: ”。
- 2) 宏定义，用于表示立即数，如len equ 100。
- 3) 数据，用于表示变量，如x dd 100。
- 4) 包含times修饰的数据，用于表示数组，如array times 100 db 0。

前面两种形式易于理解，使用如下产生式表示：

```
<lbtail>->COLON | KW_EQU NUM
```

后面两种统称为NASM格式的数据，拥有公共的尾部，差别在于是否使用times进行修饰。我们使用非终结符<basetail>表示公共的尾部，则使用如下产生式表示NASM格式的数据：

```
<lbtail>->KW_TIMES NUM <basetail> | <basetail>
```

将以上产生式合并，得到<lbtail>的产生式：

```
<lbtail>->COLON  
  
          | KW_EQU NUM  
  
          | KW_TIMES NUM <basetail>  
  
          | <basetail>
```

NASM格式的数据尾部<basetail>包含两个部分，用于描述单元内存大小的KW_DB、KW_DW、KW_DD（我们称为<len>），以及用于描述数据内容的<value>。因此<basetail>的产生式描述为：

```
<basetail>-><len> <value>  
  
<len>->KW_DB | KW_DW | KW_DD
```

其中<value>是使用逗号分割的任意多个常量（数字常量NUM和字符串常量ST）和变量（标识符ID）的重复序列，其产生式描述如下：

```
<value>-><type> <valtail>  
  
<valtail>->COMMA <type> <valtail> | ε  
<type>->NUM | <off> NUM | STR | ID  
<off>->ADD | SUB
```

非终结符<type>表示重复序列中的每一个元素，<valtail>用于表达重复的以逗号开始的子序列。<type>的产生式中，使用<off>表达数字常量的正负符号。

根据以上产生式，相信不难构造出对应的递归下降子程序。

6.2.3 指令

构造汇编指令的文法需要考虑指令的结构和操作数访问模式，结合前面章节对x86指令的格式的描述，我们可以先考虑指令的通用形式。

操作符[目的操作数][源操作数]

通用的指令形式包含操作符、目的操作数、源操作数三部分，由于我们的编译器没有生成包含指令前缀的汇编指令，因此这里不考虑指令前缀的存在。x86指令集的操作数是可选部分，因此可以将x86指令分为三大类：双操作数指令、单操作数指令和无操作数指令。使用文法描述如下。

```
<inst>-><doubleop> <oprand> COMMA <oprand>

      | <singleop> <oprand>

      | <noneop>

<doubleop>->I_MOV | I_CMP | I_SUB | I_ADD

          | I_AND | I_OR | I_LEA

<singleop>->I_CALL | I_INT | I_IMUL | I_IDIV |

          | I_NEG | I_INC | I_DEC

          | I_JMP | I_JE | I_JNE

          | I_SETE | I_SETNE | I_SETG | I_SETGE | I_SETL
```

```
| I_SETLE | I_PUSH | I_POP  
<noneop>->I_RET
```

其中<doubleop>、<singleop>、<noneop>分别表示双操作数指令操作符、单操作数指令操作符和无操作数指令操作符。

<operand>表示任意形式的操作数，其文法定义与x86指令的操作数寻址模式息息相关。在x86指令集中，操作数的来源有三种，即立即数、寄存器和内存，分别对应立即寻址、寄存器寻址和内存寻址。立即数有两种形式，即数字常量NUM和用于表示地址或者值的标识符ID。因此，<operand>文法定义为：

```
<operand>->NUM | ID | <reg> | <mem>
```

非终结符<reg>表示寄存器操作数，我们定义的汇编语言只使用了8位和32位通用寄存器，因此使用的寄存器一共是16个。

```
<reg>->BR_AL | BR_CL | BR_DL | BR_BL  
  
| BR_AH | BR_CH | BR_DH | BR_BH  
  
| DR_EAX | DR_ECX | DR_EDX | DR_EBX  
  
| DR_ESP | DR_EBP | DR_ESI | DR_EDI
```

非终结符<mem>表示内存操作数，根据前面章节对x86指令内存寻址的描述，内存操作数的一般形式为：

```
<mem>->LBRAC <addr> RBRAC
```

其中<addr>表示内存地址，内存寻址模式有直接寻址、寄存器间址、基址+偏移寻址、基址+变址寻址和基址+变址+偏移5种寻址模式。除了直接寻址模式的内存地址为立即数（或者称为偏移）之外，其他寻址模式都是以寄存器名开始的地址表达式，因此内存操作数的文法定义为：

```
<addr>->NUM | ID | <reg> <regaddr>
```

其中<regaddr>表示以寄存器名开始的地址表达式的后半部分，如果<regaddr>为空则表示寄存器间址，否则表示包含基址寄存器的寻址模式。

```
<regaddr>-><off> <regaddrtail> | ε
```

由于编译器没有生成包含基址+变址+偏移寻址的操作数，因此<regaddrtail>表示偏移或变址寄存器。

`<regaddrtail>->NUM | <reg>`

至此，汇编指令的文法构造完毕。接下来与编译器类似，我们需要为语法分析器的递归下降子程序添加语义动作解析汇编语言语法模块的语义。

6.3 符号表管理

在编译器中，为了管理高级语言中的符号，包括变量名、函数名等，需要构建符号表保存符号名与相应数据结构的对应关系。同样地，在汇编器中，仍需要实现对符号信息的按名访问。

如图6-5所示，汇编语言的符号来源有四种。

- 1) 数据：用于表示一段内存区域的起始位置，一般存在于数据段中。它可能来源于高级语言中全局变量（数组）的定义，也可能来自编译器为常量字符串生成的名字标签。
- 2) 标签：用于表示一个内存地址，一般存在于代码段中。它可能来自于高级语言中的函数名，也可能来自翻译复合语句产生的标签。
- 3) 宏：用于表示一个立即数，相当于为一个数字常量起了一个名字。我们的编译器未生成宏符号，不过由于该类型的符号在汇编语言中较为常见，我们的汇编器处理该符号。
- 4) 外部符号：用于表示引用的其他文件的数据或标签。一般表示一个外部全局变量（数组）的名字，或外部函数的名字。

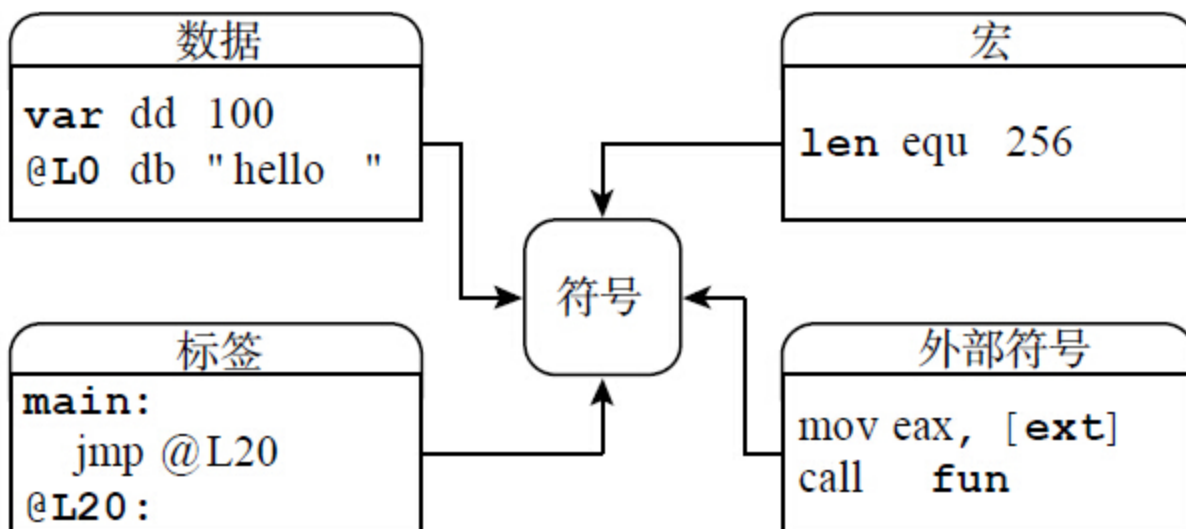


图6-5 汇编符号来源

6.3.1 数据结构

根据汇编语言符号的来源，我们定义了汇编符号的数据结构如下。

```
1 struct lb_record
{
2     static int curAddr;           //表示已分析的段的长度

3     string segName;              //符号所在段名

4     string lbName;               //符号名

5     bool isEqu;                  //是否是宏

6     bool externed;               //是否是外部符号

7     bool global;                //是否是全局符号

8     int addr;                   //符号逻辑地址

9     int times;                   //内存单元重复次数

10    int len;                     //单位内存大小

11    list<int> cont;               //符号内容

12    lb_record(string n,bool ex=false); //标签或外部符号

13    lb_record(string n,int v);      //宏符号

14    lb_record(string n,int t,int l,list<int> c); //数据符号

15    void write();                 //输出符号内容

16 };
```

1) 字段**curAddr**表示汇编器语法分析过程中，当前分析的段的长度，也就是下一个符号的起始地址（相对于段起始地址的偏移），初始值为0。汇编语言数据段中的数据和代码段中的指令都需要占用内存空间，因此在分析这些数据时，会不断地将其占用内存的大小累加到该字段。

2) 字段**segName**表示符号所在的段名。

3) 字段**lbName**表示符号的名称。

4) 字段**isEqu**表示符号是否是宏，如果是宏符号，该字段置为true。

5) 字段**externed**表示符号是否是外部符号，如果是外部符号，该字段置为true。

6) 字段**global**表示符号是否是全局符号，如果是全局符号，该字段置为true。

7) 字段**addr**表示符号的逻辑地址，如果符号是宏符号，则表示宏的值。

8) 字段**times**表示数据符号定义中内存单元的重复次数，默认值为1。

9) 字段**len**表示数据符号定义中单位内存的大小，取值为1、2、4字节，分别对应**db**、**dw**、**dd**。

10) 字段**cont**表示数据符号定义中的数据内容，汇编器将符号的数据内容按字节拆分，保存到一个整数链表内。

11) 第12行构造函数用于创建标签符号或外部符号，参数**ex**指定符号是否是外部符号。默认情况下直接使用标签名创建的符号对象都是汇编文件内定义的标签符号。

12) 第13行构造函数用于创建宏符号，参数**v**表示宏的值。

13) 第14行构造函数用于创建数据符号，参数**t**表示内存单元重复次数，参数**l**表示内存单元大小、参数**c**表示数据符号的内容。

14) 函数**write**输出标签在内存中的内容。由于只有数据符号占用内存，因此该函数只对数据符号有效。在目标文件生成章节会对该函数的实现进行描述。

明确了符号数据结构的定义，可以很容易构建汇编器的符号表数据结构。汇编器的符号表结构相对简单，本质上是一个以符号名字符串为键的散列表，值元素类型为**lb_record***。

```
1 class Table
```

```
{
```

```
//符号表
```

```
2      int hasName(string name);                                // 查询符号

3  public:
4      hash_map<string, lb_record*, string_hash> lb_map;        // 符号散列表

5      void addlb(lb_record*p_lb);                               // 添加符号

6      lb_record * getlb(string name);                           // 获取符号

7      void switchSeg();                                         // 段切换

8      void exportSyms();                                        // 输出符号

10 };
```

符号表使用**hash_map**作为内部存储数据结构。

- 1) 私有方法**hasName**用于查询符号是否存在，若符号存在则返回1。
- 2) 方法**addlb**向符号表中添加符号，若被添加的符号已经存在，需要进行特殊处理。
- 3) 方法**getlb**从符号表中取出符号，如果符号不存在，则需要进行特殊处理。
- 4) 方法**switchSeg**用于同步当前符号所在段的上下文信息。
- 5) 方法**exportSyms**将符号信息导出到**elf**可重定位目标文件。
- 6) 方法**write**用于输出数据段的符号内容。

汇编器除了为符号、符号表建立相应的数据结构外，还需要为分析的汇编指令建立对应的数据结构。

```
1 struct ModRM
2 {
3     int mod;
4     int reg;
5     int rm;
6 };
7 struct SIB
8 {
9     int scale;
10    int index;
11    int base;
12 };
13 struct Inst
14 {
15     unsigned char opcode;
16     int disp;
17     int imm32;
18     int dispLen;
19 };
20 extern ModRM modrm;
21 extern SIB sib;
22 extern Inst instr;
```

我们使用三个简单的全局结构体对象`instr`、`modrm`和`sib`记录汇编器语法分析过程中得到的指令及其相关的ModRM和SIB字段的信息。

回顾第5章对x86指令结构的描述，可以确定ModRM中，`mod`字段取值范围为0~3，`reg`字段取值范围为0~7，`rm`字段取值范围为0~7。`mod`字段初始值为-1，表示不存在ModRm字段，这是为何不使用“位域”定义ModRM结构体的原因（当然，也可以添加一个bool字段专门表达该信息）。同样地，SIB中的`scale`字段取值范围为0~3，`index`字段取值范围为0~7，`base`字段取值范围为0~7。`scale`字段初始值为-1，表示不存在SIB字段。`Inst`记录了指令中涉及的其他字段，操作码opcode

字段实际并未使用，`disp`表示偏移字段，`imm32`表示立即数字段，`dispLen`表示偏移字段的长度（取值1或4字节）。

汇编器语法分析中的语义动作的处理，大部分都是基于以上数据结构的操作。由于指令数据结构仅仅记录分析过程中指令的信息，是无状态的数据对象，因此在指令生成章节再对其相关操作详细阐述。接下来，我们重点分析符号数据结构相关操作的实现。

6.3.2 符号管理

符号管理涉及符号对象的创建、添加到符号表以及从符号表中取出符号的操作。由于不需要考虑汇编语法的正确性，因此相对于编译器的符号表管理，汇编器的符号管理相对简单。

1. 创建符号对象

在汇编语言中，本地声明的符号一般是以标识符开始的一段声明或定义。根据语法分析章节对符号定义尾部<lbtail>的描述，我们在其递归下降子程序中插入创建符号对象的语义动作。

```
1 void Parser::lbtail
(string lbName) {
2     move();
3     switch(look->tag) {
4         case KW_TIMES
:
5             match(NUM);
6             basetail(lbName, ((Num*)look)->val);
7             break;
8         case KW_EQU
:
9             match(NUM);
10            table.addlb(new lb_record(lbName, ((Num*)look)->val));
11            break;
12        case COLON
:
13            table.addlb(new lb_record(lbName));
14            break;
15        default:
16            basetail(lbName, 1);
17    }
18 }
```

递归下降子程序**lbtail**根据读入的词法记号决定不同的语义动作。如果读入词法记号**times**，则认为是形如“array times 100 db 0”的包含**times**的数据定义，因此继续读入重复次数，并将该值取出，与符号名**lbName**一起传递给**basetail**继续处理。如果读入词法记号**equ**，则认为是宏定义，将宏的值取出，创建符号对象，添加到符号表。如果读入词法记号‘:’，则认为是标签符号，则直接创建符号对象，添加到符号表。最后一种情况表示不包含**times**的一般数据定义形式，我们认为**times**的值为1，与第一种情况处理类似。

```
1 void Parser::basetail
(string lbName,int times) {
2     int l=len();
3     value(lbName,times,l);
4 }
5 int Parser::len

() {
6     move();
7     switch(look->tag) {
8         case KW_DB

: return 1;
9         case KW_DW

: return 2;
10        case KW_DD

: return 4;
11        default: return 0;
12    }
13 }
14 void Parser::value

(string lbName,int times,int len) {
15     list<int> cont;
16     type(cont,len);
17     valtail(cont,len);
18     table.addlb(new lb_record(lbName,times,len,cont));
19 }
20 void Parser::type

(list<int>& cont, int len) {
21     move();
22     switch(look->tag)
23     {
24         case NUM
```

```

:
25             cont.push_back(((Num*)look)->val)
26             break;
27         case STR
:
28             for(int i=0; i < ((Str*)look)->str.size(); i++) {
29                 cont.push_back(((Str*)look)->str[i])
30             }
31             break;
32         case ID
:
33             cont.push_back(table.getlb(((Str*)look)->str)->addr);
34             break;
35         default:
36     }
37 }
38 void Parser::valtail
(list<int>& cont, int len) {
39     if(match(COMMA
)) {
40         type(cont, len);
41         valtail(cont, len);
42     }
43 }

```

递归下降子程序**basetail**的目的是读取并记录数据定义的内容，实现稍微复杂，本质上是对**NASM**格式数据定义的处理。

首先它调用**len**获取数据单元的大小，**len**函数根据数据定义关键字**KW_DB**、**KW_DW**、**KW_DD**返回数据内存单元的大小**1**、**2**、**4**字节。然后将符号名、数据内容重复次数**times**以及内存单元大小传递给**value**函数继续处理。

value函数创建**list<int>**类型的对象**cont**，用于记录数据定义的内容。其中**type**函数处理各种类型的数据内容，包括数字、字符串和标识符地址。**valtail**函数处理逗号分隔的多个数据内容的情况。

在`type`函数中，如果读入词法记号为数字，则将数字的值取出放入`cont`。如果读入的词法记号为字符串，则将字符串内的字符按顺序依次放入`cont`。如果读入的词法记号为标识符，则先从符号表中取出该符号对象，然后将符号的地址放入`cont`（这种情况会涉及符号不存在的情况，稍后会描述）。

最后`value`使用处理后的`cont`，与符号名、数据重复次数以及内存单元大小，创建数据定义符号对象，并添加到符号表。

2. 添加符号对象

从前面描述的内容可知，每次创建符号对象时都会调用`addlb`把符号添加到符号表。添加符号的方法需要进行特殊处理，这是因为汇编语言符号定义和使用的特殊性——汇编语言允许符号的后置定义。例如汇编语句：

```
    jmp @L0
@L0:
```

指令`jmp`的目标标签`@L0`的定义可以出现在该指令之后，这样就带来一个问题。由于汇编器的词法分析器是按序扫描源程序的，当处理到`jmp`指令时，会从符号表内查询符号`@L0`的信息，显然此时符号表内并无该符号的信息，汇编器会将符号`@L0`作为外部符号对待。而当词法分析器扫描到符号`@L0`的定义时，才将符号`@L0`的定义信息添

加到符号表，显然符号@L0并非外部符号。类似的情况还会以如下方式出现：

```
section .text
    mov [var], 100
section .data
    var dd 0
```

一般在汇编程序中，会出现数据段“.data”定义在代码段“.text”之后的情况，那么在汇编器处理代码段中的指令时，所有指令引用的数据段的符号都不会在符号表内存在。当然，我们的编译器在生成汇编代码时“刻意”将数据段放在了代码段的前面，但仍不能避免符号定义出现在符号使用之后的情况。

解决该问题的办法是汇编器需要对汇编程序进行两遍扫描。第一次扫描时，汇编器将引用的不在符号表内的符号作为外部符号进行处理，添加到符号表。而当扫描到符号@L0的定义时，使用该符号的定义信息替换原有的符号信息。到第二次扫描时，可以确定所有汇编程序内定义的符号信息都保存到了符号表，当再次遇到使用符号的指令时便可以从符号表内查询出该符号的“真正”信息，而那些仍为外部符号的符号表项可以确定是真正的外部符号，即不在汇编程序内定义。

```
1  int scanLop = 0;                                     //扫描次数

2  void Parser::analyze
   () {
3      if(++scanLop <= 2) {                             //两遍扫描
```

```
4             move();                                //读入词法记号

5             program();                             //语法分析

6         analyze();
7     }
8 }
```

实现汇编程序的两遍扫描很简单，我们使用一个全局变量`scanLop`记录扫描的次数。只要扫描次数不大于两次，便一直调用`program`递归下降子程序重复进行语法分析。

每一遍语法分析过程中，都会调用`addlb`添加符号到符号表。为了避免添加重复的符号，因此需要对其实现进行特殊处理。

```
1 void Table::addlb
  (lb_record* p_lb) {                                //添加符号

2     if(scanLop != 1) {                             //只在第一遍添加新
符号

3         delete p_lb;
4         return;
5     }
6     if(hasName(p_lb->lbName)) {                     //本地符号覆盖外部符号

7         delete lb_map[p_lb->lbName];
8         lb_map[p_lb->lbName]=p_lb;
9     } else {
10        lb_map[p_lb->lbName]=p_lb;
11    }
12    if(p_lb->times!=0&&p_lb->segName=="data") {
13        defLbs.push_back(p_lb);                     //记录包含数据的符
符号

14    }
15 }
```

在函数addlb中，我们只需要在第一次扫描时向符号表内添加符号即可，如果scanLop不等于1则立即返回。如果在添加符号时发现该符号已经存在于符号表，则断定符号表内保存的必然是由getlb添加的外部符号（getlb函数在找不到符号时，会自动创建一个外部符号添加到符号表），这是因为我们编译的合法汇编程序是不会出现符号重定义的。由于被添加的符号一定是本地定义的（externed字段为false），所以直接将该符号的信息更新到符号表即可。最后，我们将数据段“data”内的times不为0（包含数据）的符号按序记录到defLbs中，以供后续生成数据段内容时使用。

3.获取符号对象

获取符号对象的方法getlb的实现与addlb是相关的。

```
1 lb_record * Table::getlb
  (string name) {
2     lb_record* ret;
3     if(hasName(name)) {                                     //符号存在

4         ret=lb_map[name];
5     } else {
6         ret = new lb_record(name, true);                   //创建外部符号

7         lb_map[name] = ret;                                 //添加到符号表

8     }
9     return ret;
10 }
```

在函数`getlb`中，第一遍扫描时，如果符号存在于符号表，则直接返回。否则将创建一个外部符号添加到符号表内，这是汇编器最后一种创建符号的情况。当第二遍扫描时，无论是什么类型的符号，总能在符号表内找到。而且我们可以确定，对于先引用后定义的本地符号，`addlb`已经在第一遍处理中将符号表内的记录更新为符号的真正定义。

获取符号的语义动作出现于以下四种情况：

- 1) 形如： `global x` 。
- 2) 形如： `ptr dd x` 。
- 3) 形如： `mov eax, x` 。
- 4) 形如： `mov eax, [x]` 。

第一种情况声明`x`为全局符号，此时需要取出`x`的符号对象，将`isGlb`设置为`true`。第二种情况表示数据定义中引用了符号的地址作为数据内容，这个情况已经在“创建符号对象”章节的`type`递归下降子程序中描述了。后面两种情况是指令使用符号地址作为立即数或者内存地址（如果符号是宏符号，则表示其对应的值），其调用代码会在“指令生成”章节详细描述。

6.4 表信息生成

汇编器最终输出ELF格式的可重定位目标文件。第5章描述了ELF文件的通用结构，而在汇编器中我们只关心可重定位目标文件涉及的ELF文件结构。

文件头 (ELF Header)	52
代码段 (.text)	
数据段 (.data)	
段表字符串表 (.shstrtab)	
段表 (Section Header Table)	(40*段表项个数)
符号表 (.symtab)	(16*符号表项个数)
字符串表 (.strtab)	
重定位表 (.rel .text)	(8*重定位表项个数)
重定位表 (.rel .data)	(8*重定位表项个数)

图6-6 可重定位目标文件结构

如图6-6所示，在可重定位目标文件中，不会包含程序头表。另外编译器生成的代码不包含“.bss”段，因此只需要考虑代码段和数据段。其中，代码段的生成在指令生成章节再详细描述，数据段的内容来源于符号表内的数据定义标签。另外，ELF文件头、段表字符串表、字

字符串表在ELF文件结构确定后可以计算得出。因此，在可重定位目标文件中最关键的三个段是：段表、符号表和重定位表，这三个表是表信息生成的主要内容。

汇编器在第一遍扫描时，将汇编文件的段信息导出为段表项，填充到段表。第二遍扫描时，将符号信息导出为符号表项，填充到符号表，并在产生重定位的位置生成重定位项，填充重定位表。

6.4.1 段表信息

汇编语言使用**section**关键字声明段起始位置，下一个段声明或文件结束位置为段终止位置，中间部分属于**section**声明的段内容。因此在汇编器语法分析过程中，可以在**section**声明或文件结束时处理段信息。

```
1 void Parser::program()
2 {
3     switch(look->tag){
4         case END:                                     //文件结
束, 停止语法分析

5             table.switchSeg

6             return;
7         case KW_SEC:                                 //段声明

8             match(ID);
9             table.switchSeg

10            break;
11        case KW_GLB:                                 //全局符号
声明

12            match(ID);
13            break;
14        case ID:                                     //数据定义

15            lbtail();
15            break;
16        default:                                     //指令

17            inst();
18    }
19    program();
20 }
```

函数switchSeg在段声明位置和文件结束位置处理段信息。ELF文件段表项中最关键的字段是段名、段基址和段大小。由于汇编语言段内包含的数据定义或汇编指令的大小是可以确定的，因此通过第一遍扫描可以确定段表项内的信息。其中，段表名由section关键字声明的标识符决定，段表大小由段内的数据大小决定，段表的偏移由上个段结束位置的对齐位置决定（默认情况下，段偏移的默认按照4字节大小对齐）。

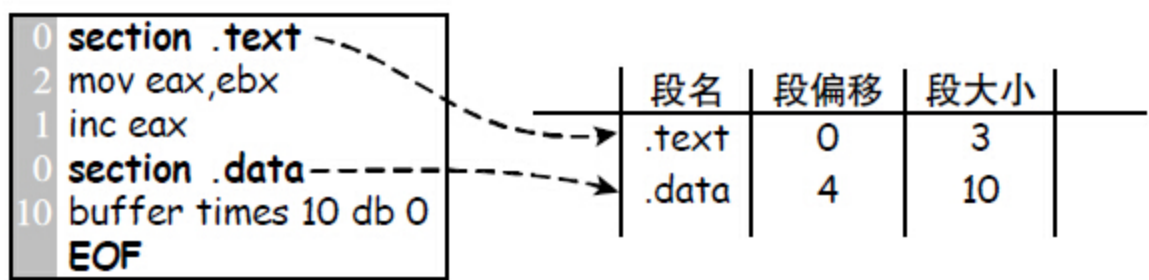


图6-7 段表信息生成

如图6-7所示，代码段“.text”段偏移默认从0字节开始，段内含有两条汇编指令，mov指令的一种二进制编码为0x8bc3（参考5.1节的内容），大小为2字节，inc指令的二进制编码为0x40，大小为1字节，因此“.text”段大小为3字节。数据段“.data”段偏移从“.text”结束位置3字节开始，按照4字节对齐后为4字节，“.data”段内定义了10字节的buffer，因此“.data”段大小为10字节。

```
1 int dataLen=0;
2 string curSeg="";
3 void Table::switchSeg
() {
```



```

8 public:
9     Elf32_Ehdr ehdr;                                     //文件头

10     vector<Elf32_Phdr*>phdrTab;                           //程序头表

11     hash_map<string, Elf32_Shdr*, string_hash> shdrTab;    //段表

12     vector<string>shdrNames;                             //段名顺序

13     hash_map<string,Elf32_Sym*,string_hash>symTab;        //符号表

14     vector<string>symNames;                              //符号名顺序

15     vector<RelItem*>relTab;                              //重定位表

16     string shstrtab;                                     //段表字符
串表

17     string strtab;                                     //字符串表

```

其中RelItem类是对重定位表项的简单封装，ELF_file表示整个ELF文件，Elf32_*表示所有ELF文件结构类型（定义见Linux系统的“/usr/include/elf.h”文件）。addShdr函数便是对shdrTab字段的操作。

```

1 void Elf_file::addShdr
(string sh_name,int size) {
2     int off=size of (Elf32._Ehdr)+dataLen;
3     if(sh_name==".text") {
4         addShdr(sh_name,SHT_PROGBITS,
5                 SHF_ALLOC|SHF_EXECINSTR,
6                 0,off,size,0,0,4,0);
7     }
8     else if(sh_name==".data") {
9         addShdr(sh_name,SHT_PROGBITS,
10                SHF_ALLOC|SHF_WRITE,
11                0,off,size,0,0,4,0);
12     }
13 }
14
15 void Elf_file::addShdr
(
16     string sh_name,
17     Elf32_Word sh_type,
18     Elf32_Word sh_flags,

```

```

19         Elf32_Addr sh_addr,
20         Elf32_Off  sh_offset,
21         Elf32_Word sh_size,
22         Elf32_Word sh_link,
23         Elf32_Word sh_info,
24         Elf32_Word sh_addralign,
25         Elf32_Word sh_entsize
26     ) {
27         Elf32_Shdr*sh=new Elf32_Shdr();
28         sh->sh_name=0;
29         sh->sh_type=sh_type;
30         sh->sh_flags=sh_flags;
31         sh->sh_addr=sh_addr;
32         sh->sh_offset=sh_offset;
33         sh->sh_size=sh_size;
34         sh->sh_link=sh_link;
35         sh->sh_info=sh_info;
36         sh->sh_addralign=sh_addralign;
37         sh->sh_entsize=sh_entsize;
38         shdrTab[sh_name]=sh;
39         shdrNames.push_back(sh_name);
40     }

```

第1~13行定义的函数addShdr根据段名决定段表项其他属性的值。比如第3~7行处理代码段“.text”时，将段属性设置为SHT_PROGBITS表示该段保存了程序数据，设置为SHF_ALLOC表示段加载时需要分配内存空间，设置为SHF_EXECINSTR表示该段具有可执行权限。

第15~40行定义的函数addShdr构造Elf32_Shdr对象，将该对象保存到段表shdrTab，并记录段表名的顺序到shdrNames。另外，段表项的sh_name字段暂时初始化为0，是因为目前还未添加段表字符串表段“.shstrtab”，段表名在该段内的偏移尚未确定，只有在组装可重定位目标文件时才能最终确定sh_name的值。

6.4.2 符号表信息

这里所说的符号表是ELF文件的符号表，而非6.3节所描述的符号表数据结构，不过这二者之间是有关联的。符号表数据结构记录了汇编代码中所有的符号信息，包括数据定义符号、宏符号、代码标签、函数名等。而ELF文件的符号表保存的符号信息是为链接器、调试器、反汇编器等服务的。对于我们后面实现的链接器来说，其实只需要关心全局符号即可。不过为了尽可能保持ELF符号表信息的完整性，我们也将局部符号保存到ELF符号表内。因此，可以简单认为ELF文件的符号表是汇编器符号表的一个子集，只不过前者是在二进制层面描述符号，后者是在汇编语言层面描述符号。

由于汇编器在第二遍扫描时已经将符号的信息全部收集到符号表内，因此我们只需要稍加处理，便可以得到ELF文件的符号表。ELF文件符号表项中最关键的字段是符号名、符号所在段、符号段内偏移和符号类型（全局/局部）。

如图6-8所示，符号main被扫描时可以从全局变量curSeg中取出当前段的名称“.text”，从lb_record: : curAddr取出符号在段内的偏移地址0，并根据global声明将main符号设置为全局符号。而对于符号whileExit1是在第二遍扫描开始时才确定是本地符号的，它在“.text”段内的偏移地址是22。另外，符号fun是引用的外部符号，不能确定它所

在的段。而对于外部符号，我们统一设定为全局符号，这是因为链接器会对全局符号进行符号解析，而忽略局部符号的内容。

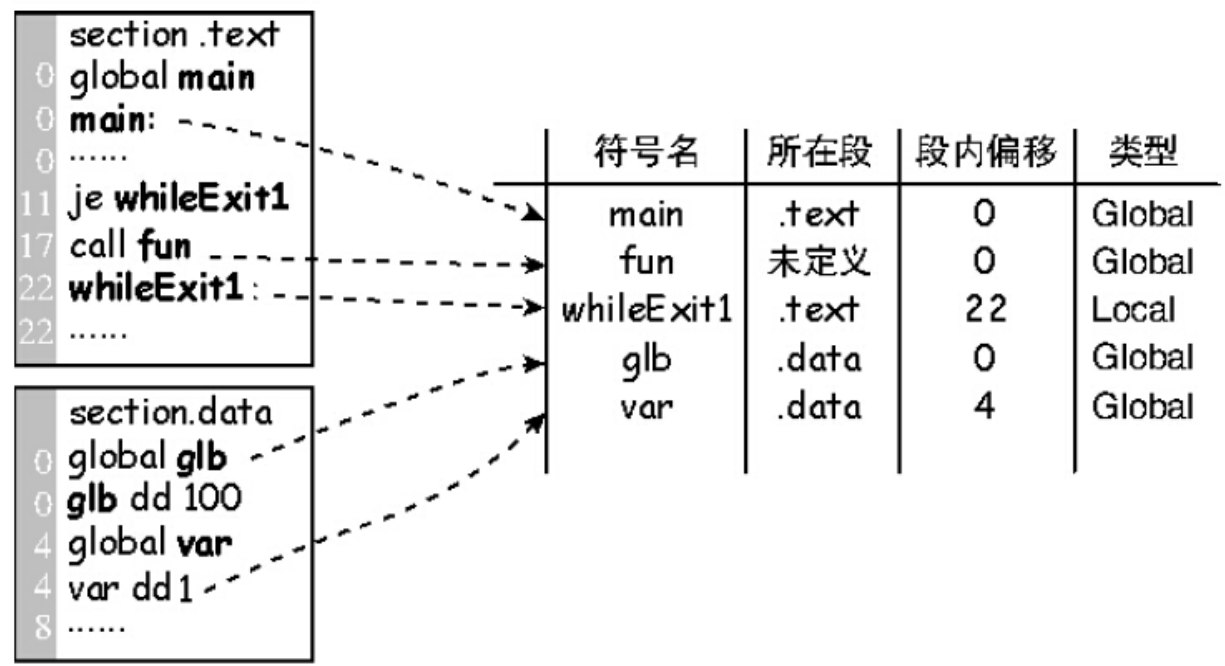


图6-8 符号表信息生成

```
1 void Table::exportSyms
2 {
3     for(hash_map<string, lb_record*, string_hash>
4         ::iterator lb_i=lb_map.begin();
5         lb_i!=lb_map.end();lb_i++) {
6         lb_record *lr=lb_i->second;
7         if(!lr->isEqu)
8             obj.addSym(lr);
9     }
```

汇编器两遍扫描结束后，会调用exportSyms将符号导出到ELF符号表。由于宏符号并不是有意义的符号，因此不会被导出。addSym函数会根据符号对象的内容构造ELF符号表项Elf32_Sym，添加到符号表symTab。

```

1 void Elf_file::addSym
  (lb_record*lb) {
2     Elf32_Sym*sym=new Elf32_Sym();
3     sym->st_name=0;                                     //
符号名

4     sym->st_value=lb->addr;                             //符号
地址

5     sym->st_size=lb->times*lb->len*lb->cont.size();      //符号大小

6     if(lb->gloabl) {                                     //
全局符号

7         sym->st_info=ELF32_ST_INFO(STB_GLOBAL,STT_NOTYPE);
8     } else {
//局部符号

9         sym->st_info=ELF32_ST_INFO(STB_LOCAL,STT_NOTYPE);
10    }
11    sym->
st_other=0;
12    if(lb->externed) {                                     //
外部符号

13        sym->st_shndx=STN_UNDEF;
14    } else {
//本地符号

15        sym->st_shndx=getSegIndex(lb->segName);
16    }
17    symTab[lb->lbName]=sym;
18    symNames.push_back(lb->lbName);
19 }

```

在函数addSym中，对Elf32_Sym对象的设置如下：

第3行，将st_name设置为0，这是因为符号表依赖的字符串表“.strtab”还未添加，只有在组装可重定位目标文件时才能最终确定st_name的值。

第4行，将`st_value`设置为符号地址`addr`，即符号的段内偏移地址，对于外部符号，该值默认为0。

第5行，计算符号大小`st_size`，即`times`、`len`与符号内容`cont`长度的乘积。

第6~10行，处理符号的全局/局部类型。我们并不关心`st_info`的`TYPE`字段，因此默认设置为`STT_NOTYPE`。对于全局符号，`st_info`的`BIND`字段值为`STB_GLOBAL`，对于局部符号，`st_info`的`BIND`字段值为`STB_LOCAL`。最后，使用`ELF32_ST_INFO`宏将`TYPE`与`BIND`字段合并为`st_info`的最终值。

第12~16行处理外部/本地符号。对于外部符号，符号所在段对应段表项的索引`st_shndx`是不确定的，因此设置为`STN_UNDEF`。而对于本地符号，则取出符号所在段名`segName`，并使用`getSegIndex`函数获取该段名在段名列表`segNames`中的索引，设置为`st_shndx`的值。

最后，将符号保存到符号表`symTab`，并将符号名插入符号名列表`symNames`。

6.4.3 重定位表信息

目标文件的重定位表是汇编器与链接器关系中最重要的一环，链接器正是读取目标文件的重定位表信息对目标文件进行链接的工作。前面在介绍目标文件符号表生成的时候，每个符号表项存储的符号的值都是符号在所在段内的偏移地址，而非真实的虚拟地址，而链接器会将这些偏移地址转换为虚拟地址。目标文件的符号地址不是真实虚拟地址将会带来一个问题，那些引用该符号的数据定义或指令的内容并不是有效的。正因如此，汇编器需要将这些信息收集到重定位表中，告诉链接器哪些数据或指令需要按照怎样的方式进行修正。

前面介绍过汇编语言内的符号大致分为数据、标签、宏以及外部符号四大类，而本质上讲，汇编语言的符号其实就两种，即表示数据地址的符号和表示指令地址的符号，宏符号不会出现在目标文件内，而外部符号无外乎数据和标签两种符号。对于符号的引用者来说，存在两种场景：本地引用，即引用符号的位置和符号定义位置在同一个文件的同一段内；外部引用，即引用符号的位置和符号定义位置不在同一文件内或在同一文件的不同段内。之所以强调是否同一段内而非同一文件内，是因为链接器工作时会调整所有段的位置，而不管这些段是否在同一个文件内。

在汇编代码中所有出现的对数据和标签符号的引用都有可能需要进行重定位，根据符号类型和引用场景可以分为四种情况进行考虑：

1) **本地引用数据符号**：这种情况常见的场景是全局变量定义“char*str="hello"; ”生成的汇编代码。

```
@L0 db "hello"  
str dd @L0
```

虽然符号“@L0”和“str”在同一个“.data”段内，但是“str”存储的是“@L0”的偏移地址，链接器处理后，“@L0”的地址被修正为虚拟地址，而“str”存储的数据内容也需要改变为“@L0”对应的虚拟地址，因此“str”的数据内容需要被重定位。

2) **外部引用数据符号**：这种情况常见的场景是对全局变量的访问，比如“x=1; ”生成的汇编代码。

```
mov [x], 1
```

与情况1类似，链接器处理后，“x”的地址被修正为虚拟地址，那么mov指令内保存的x的地址需要更正为对应的虚拟地址，因此“mov”指令的内容需要重定位。不管x是否是本地文件的全局变量还是通过“extern”声明的外部变量，都需要这样的处理。

3) **本地引用标签符号**：这种情况常见的场景是调用本地声明的函数，比如“`fun () ;`”生成的汇编代码。

```
call fun
fun:
```

我们知道，“`call`”指令的内容保存的是被调用函数的地址相对于“`call`”指令的下一条指令的地址的偏移，假设“`fun`”的定义就在“`call`”指令之后，那么“`call`”指令内保存的相对地址就是0。链接器处理后，“`fun`”的地址被修正为虚拟地址。然而“`call`”指令和“`fun`”符号定义都是在“`.text`”段内，无论链接器如何调整“`.text`”的位置，“`call`”指令和“`fun`”的符号地址的相对位置都不会发生改变，即“`call`”指令的内容不会发生变化，也就无须对“`call`”指令进行重定位。类似的情况还会发生在函数内部复合语句生成的跳转指令`jmp/Jcc`指令中，这些指令也是不需要重定位的。

4) **外部引用标签符号**：这种情况常见的场景是调用外部文件声明的函数，还是“`fun () ;`”生成的汇编代码。

```
call fun
```

只不过这次“`fun`”符号是未知符号，虽然我们可以推测该符号也是在“`.text`”段内，但是它并不是本地的“`.text`”段。链接器处理时会把同名的段进行合并，但是即便如此，我们也无法计算“`call`”指令与“`fun`”符号

地址的相对位置。因此，链接器处理后，“call”指令内的相对地址需要根据“fun”的虚拟地址重新定位。

根据以上分析，我们可以得出结论：凡是对数据符号的引用都需要重定位，而对标签符号的引用，只有外部引用时才需要重定位。

```
1 void Parser::type
(list<int>& cont, int len) {
2     move();
3     switch(look->tag) {
4         case NUM:
5             cont.push_back(((Num*)look)->val)
6             break;
7         case STR:
8             for(int i=0; i < ((Str*)look)->str.size(); i++) {
9                 cont.push_back(((Str*)look)->str[i])
10            }
11            break;
12        case ID:
13            if(scanLop==2 && !lr->isEqu) {
14                obj.addRel
15
16                (
17                    curSeg, lb_record::curAddr+cont.size()*len,
18                    name, R_386_32);
19            }
20            cont.push_back(table.getlb(((Str*)look)->str)->addr);
21            break;
22        default:
23    }
```

递归下降子程序type处理数据符号的值，当它扫描到数据符号的值是ID时，如果当前是汇编器的第二遍扫描，且扫描到的符号不是宏符号，便调用addRel将生成的重定位项添加到重定位表。判断是否是第二遍扫描是保证所有的符号信息已经保存到汇编器符号表内，而判断是否是宏符号则是宏符号的值会被直接写入数据符号的内容，不存在对符号的引用，因而不需要重定位。

处理完数据段内符号引用情况，接下来处理代码段内的符号引用。我们定义的汇编语言的代码段内对符号的引用有两种情况，即将符号的值作为立即数或内存地址。下面分别讨论这两种情况。

```
1  int Parser::getRegCode
   (Tag reg, int len) {
2      int code = 0;
3      switch (len) {
4          case 4:
5              code = reg - BR_AL;
6              break;
7          case 8:
8              code = reg - DR_EAX;
9              break;
10         }
11     return code;
12 }
13
14 lb_record*relLb
   =NULL;                                // 重定位的标签

15
16 void Parser::operand
   (int &regNum,int&type,int&len) {
17     move();
18     switch(look->tag) {
19         case NUM:                                // 立即数
20             type=IMMEDIATE;
21             instr.imm32=((Num*)look)->val;
22             break;
23         case ID:                                // 立即数
24             type=IMMEDIATE;
25             string name=((Id*)look)->name;
26             lb_record* lr=table.getlb(name);
27             instr.imm32=lr->addr;
28             if(scanLop==2 && !lr->isEqu) {
29                 relLb=lr;
30             }
31             break;
32         case LBRAC:                                // 内存寻址
33             type=MEMORY;
34             mem();
35             break;
36         case SUB:                                // 负立即数
```

```

37     type=IMMEDIATE;
38     match(NUM);
39     instr.imm32=-((Num*)look)->val;
40     break;
41 default:                                     //寄存器操
作数

42     type=REGISTER;
43     len=reg();
44     int regCode = getRegCode(look->tag, len);    //寄存器编码

45     if(regNum++!=0) {                         //双寄存器
操作数

46         modrm.mod=3;
47         modrm.rm=regCode;                     //源寄存器
存入

rm
48     } else {
49         modrm.reg=regCode;                     //目标寄存
器操作数存入

reg
50     }
51 }
52 }

```

递归下降子程序operand处理汇编指令的操作数，参数regNum记录指令中已经识别的寄存器操作数的个数，type记录操作数的类型（立即数操作数—IMMEDIATE、寄存器操作数—REGISTER、内存操作数—MEMORY），len记录操作数的长度（8位或32位）。

1) 第19~22行处理数字常量立即数操作数。首先将操作数类型设为IMMEDIATE，然后取出数字常量的值存入指令的imm32字段。

2) 第23~31行处理符号立即数操作数。首先将操作数类型设为IMMEDIATE，然后根据符号名从符号表中取出符号对象，将符号地址写入指令的imm32字段。由于该处是对符号的引用，因此需要产生重定

位项。我们将符号对象记录到全局变量relLb中，指令生成阶段会读取该变量的值，进行重定位项的处理。

3) 第32~35行处理内存操作数。首先将操作数类型设为MEMORY，然后调用mem子程序处理内存操作数的情况。

4) 第36~40行处理负数立即操作数。

5) 第41~50行处理寄存器操作数。首先将操作数类型设置为REGISTER，然后调用getRegCode计算寄存器在指令中的编码。当第一次识别到寄存器操作数时，将寄存器编码记录到指令的reg字段内。第二次识别到寄存器操作数时，将指令的mod字段设为3表示指令是双寄存器操作数指令，将指令的rm字段设置为第二次识别寄存器的编码。之所以这样设置，是因为我们在指令生成时使用的双寄存器操作数指令的形式为reg字段作为目标寄存器，rm字段作为源寄存器。

```
1 void Parser::addr
() {
2     move();
3     switch(token) {
4         case number: //直
接寻址

5         modrm.mod=0;
6         modrm.rm=5;
7         instr.setDisp((Num*)look)->val,4);
8         break;
9         case ident: //直
接寻址

10        modrm.mod=0;
11        modrm.rm=5;
12        name=((Id*)look)->name;
13        lb_record* lr=table.getlb(name);
14        instr.setDisp(lr->addr,4);
```

```

15             if(scanLop==2 && !lr->isEqu) {
16                 relLb=lr;

17             }
18             break;
19         default:                                     //寄
寄存器寻址

20             int type=reg();
21             regaddr(token,type);
22         }
23     }

```

递归下降子程序**addr**处理内存操作数的内存地址。

1) 第4~8行处理数字常量的内存地址。首先将指令的**mod**字段设为0，**rm**字段设为5，表示操作数使用32位偏移直接寻址。调用指令的**setDisp**函数保存内存地址的值和长度。

2) 第9~18行处理符号内存地址。与数字常量内存地址处理方式类似，仍是设置指令的**mod**字段为0，**rm**字段为5。然后根据符号名从符号表内取出符号对象，将符号对象的地址和地址长度设置为指令内。由于该处是对符号的引用，因此需要产生重定位项。我们将符号对象记录到全局变量**relLb**中，指令生成阶段会读取该变量的值，进行重定位项的处理。

3) 第19~21行处理寄存器寻址的情况。

```

1  bool Generator::processRel
(int type) {
2      if(scanLop==1||relLb==NULL)      {
3          relLb=NULL;
4          return false;
5      }
6      bool flag=false;

```

```

7          if(type==R_386_32
) {                                     //绝对地址重定位

8          obj.addRel
(curSeg,lb_record::curAddr,
9              relLb->lbName,type);
10             flag=true;
11         }
12     else if(type==R_386_PC32
) {                                     //相对地址重定位

13         if(relLb->externed) {         //外部跳转

14             obj.addRel
(curSeg,lb_record::curAddr,
15                 relLb->lbName,type);
16                 flag=true;
17             }
18         }
19         relLb=NULL;
20         return flag;
21 }

```

在指令生成阶段，会调用`processRel`处理可能的重定位项。参数`type`表示调用者传入的重定位类型，对于一般的直接引用符号地址的指令会传入`R_386_32`表示绝对地址重定位，而对于函数调用或者跳转指令会传入`R_386_PC32`表示相对地址重定位。

1) 第2~5行表示只在第二遍扫描时进行重定位项的生成，并且要求`relLb`不能为空。`relLb`记录了指令中需要重定位的符号对象，由于我们的编译器不会生成引用多个符号的指令，因此使用一个`relLb`变量记录是一种简化的做法。

2) 第6~11行处理绝对地址重定位。通过调用`addRel`函数，将当前段名（被重定位的段）、当前地址（被重定位位置的段内偏移）、重

定位符号名（重定位的符号）和重定位类型传入，生成对应的重定位表项。

3) 第12~18行处理相对地址重定位。首先判断符号`relLb`是否是外部符号，对于本地符号则不需要生成重定位项。因为如果符号是数据段内的符号，我们处理的汇编代码不包含对数据段内符号的相对地址引用。如果符号是代码段内的符号，则表示符号和符号的引用位置在同一个段内，根据前面对符号引用的讨论，不需要生成重定位项。最后，仍调用`addRel`函数生成重定位项。

```
1 RelItem* Elf_file::addRel
  (string seg,
2   int addr,string lb,int type) {
3   RelItem*rel=new RelItem(seg,addr,lb,type);
4   relTab.push_back(rel);
5   return rel;
6 }
```

函数`addRel`的实现很简单，即根据传入的重定位段、重定位地址、重定位符号和重定位类型信息生成重定位项对象`RelItem`，然后存入重定位表`relTab`即可。其中`RelItem`的`rel`字段为`Elf32-Rel`类型，该类型的`r-offset`设为`addr`的值，`r-info`字段设为`ELF32-R-INFO (O, type)`，即只保存重定位类型信息。在目标文件生成阶段会自动填充`r-info`字段中的重定位符号信息，再将该对象的信息输出为`ELF`文件格式的重定位段的内容。之所以这样做的原因是重定位项内涉及了段名和符号名的引用，只有段表和符号表生成后才能确定这些字段在重定位表内的值。

为了更好地理解重定位表生成的流程，我们使用一个简单的例子说明这一点，如图6-9所示。

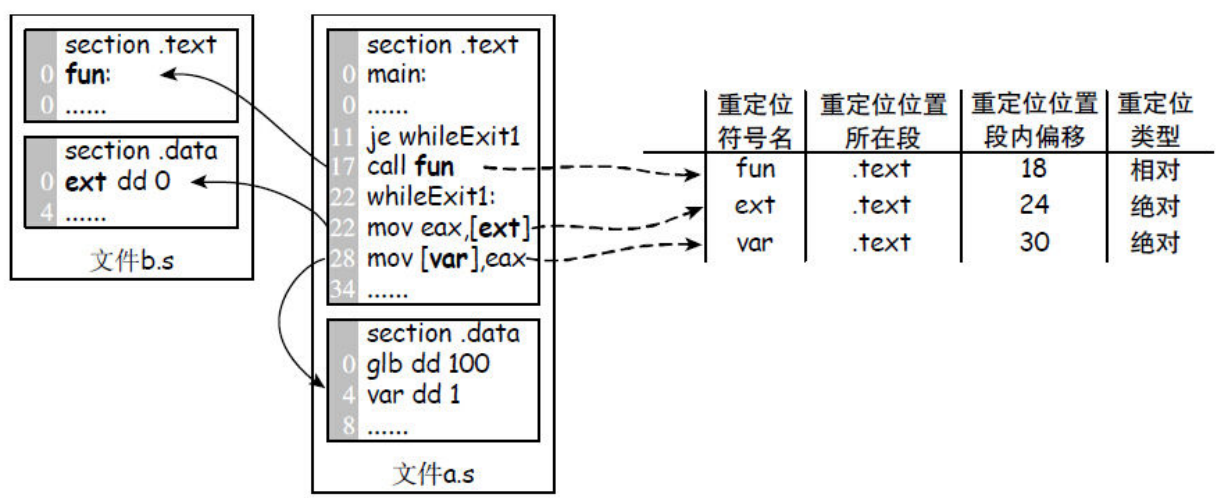


图6-9 重定位表信息生成

为了简化代码结构，我们省略了全局符号的global声明指令。图6-9中展示了前面讨论的重定位信息的来源。

1) 同文件段间符号引用最典型的是文件内代码段的指令引用了文件内数据段的符号，比如a.s的代码段的mov[var], eax指令引用了a.s的数据段内符号var。汇编器处理该指令时，var符号地址的段内偏移是30（mov指令操作码0x88占用1字节，ModR/M字段0x05占用1字节），于是根据重定位段“.text”、重定位位置30、重定位符号“var”和重定位类型生成重定位项。数据段内的符号引用情形类似，不再赘述。

2) 一个文件的代码段的指令引用了另一个文件数据段的符号，比如文件a.s代码段的mov eax, [ext]指令引用了文件b.s数据段定义的全局

符号`ext`。汇编器处理该指令时，`ext`符号地址的段内偏移是24（`mov`指令操作码`0x8a`占用1字节，`ModR/M`字段`0x05`占用1字节），于是根据重定位段“`.text`”、重定位位置24、重定位符号“`ext`”和重定位类型生成重定位项。

3) 一个文件的代码段的指令引用了另一个文件代码段的符号，比如文件`a.s`代码段的`call fun`引用了文件`b.s`代码段定义的全局符号`fun`。汇编器处理该指令时，`fun`符号相对地址的段内偏移是18（`call`指令操作码`0xe8`占用1字节），于是根据重定位段“`.text`”、重定位位置18、重定位符号“`fun`”和重定位类型生成重定位项。

4) 最后，同文件段内符号相对引用不会产生重定位项，比如`je whileExit1`对`whileExit1`局部符号的引用，因为`je`指令和`whileExit1`的相对地址永远是固定的5字节。

经过前面的讨论，我们明确了ELF文件内最关键的三个表数据结构段表、符号表、重定位表的构造方式。根据这三个表结构，我们可以构造出ELF可重定位目标文件的“骨架”。不过在汇编器中，除了构造目标文件的结构信息，还有一个重要的功能尚未讨论，即根据第5章对x86指令结构的描述将汇编代码翻译为二进制代码，这便是“指令生成”章节需要讨论的内容。

6.5 指令生成

在“语法分析”章节中，我们描述了汇编语言指令的文法。我们将待处理的汇编指令分为三类：双操作数指令、单操作数指令和零操作数指令。其中双操作指令包含mov、cmp、add、sub、and、or和lea共7种指令，单操作数指令包含call、int、imul、idiv、neg、inc、dec、jmp、je、jne、sete、setne、setg、setge、setl、setle、push和pop共18种指令，无操作数指令包含ret共1种指令。指令生成的目的便是将这些指令翻译为它们的二进制表示。

汇编器语法分析器的inst递归下降子程序识别所有的指令，然后将指令的信息记录下来，最后将指令的信息交给指令生成器转化为二进制代码。

```
1  enum op_type
   {
2      NONE,
3      IMMEDIATE,
4      REGISTER,
5      MEMORY
6  }
7
8  void Parser::inst
   () {
9      instr.init
10
11      move();
12      int len=0;
13  }
```

//操作数类型

//初始化指令信息

//

操作数长度

```
12          if(look->tag>=I_MOV
&&look->tag<=I_LEA
) {          //双操作数指令

13          op_type d_type=NONE, s_type=NONE;          //
操作数类型

14          int regNum=0;          //
寄存器个数

15          operand(regNum, d_type, len);
16          match(COMMA);
17          operand(regNum, s_type, len);
18          generator.gen2op
(look->tag, d_type, s_type, len);
19      }else if(look->tag>=I_CALL
&&look->tag<=I_POP
){          //单操作数指令

20          op_type type=NONE, regNum=0;
21          operand(regNum, type, len);
22          generator.gen1op
(look->tag, type, len);
23      }else if(look->tag==I_RET
) {          //零操作数指令

24          generator.gen0op
(look->tag);
25      }
26 }
```

1) 第9行对指令对象instr初始化，其中比较关键的是将modrm的mod和sib的scale字段初始化为-1，表示未初始化状态。

2) 第12~18行处理双操作数指令。根据词法记号标签的定义，在I_MOV和I_LEA之间的标签之间的都是双操作数指令。当子程序operand识别指令的两个操作数后，会将操作数的信息保存在instr、

modrm和sib字段内。最后调用指令的gen2op函数生成双操作数指令的二进制代码。

3) 第19~22行处理单操作数指令。根据词法记号标签的定义, 在I_CALL和I_POP之间的标签之间的都是单操作数指令。当子程序operand识别指令的唯一的操作数后, 会将操作数的信息保存在instr、modrm和sib字段内。最后调用指令的gen1op函数生成单操作数指令的二进制代码。

4) 第23~25行处理零操作数指令。I_RET是唯一的零操作数指令, 通过调用gen0op函数生成零操作数指令的二进制代码。

接下来, 我们详细描述指令生成器对每种指令的处理细节。

6.5.1 双操作数指令

在我们实现的汇编器中，只需要处理mov、cmp、add、sub、and、or和lea这7种双操作数指令即可。这些指令有很大的相似之处，除了lea指令稍微特殊外，其他指令的操作数几乎可以是任何类型的合法操作数，即目的操作数可以是寄存器操作数、内存操作数，源操作数可以是立即数、寄存器操作数和内存操作数。另外，考虑到指令的操作数的长度可以是8位或32位（不考虑16位操作数），因此每种双操作数指令都有很多的组合。不过我们只考虑以下操作数的组合（我们使用OP表示任意的操作码）。

```
1)OP reg8, reg8
2)OP reg8, mem8
3)OP mem8, reg8
4)OP reg8, imm8
5)OP reg32, reg32
6)OP reg32, mem32
7)OP mem32, reg32
8)OP reg32, imm32
```

其中组合1、2我们统一使用指令“OP rm8, reg8”生成，组合5、6统一使用指令“OP rm32, reg32”生成。最后，对于包含立即数的指令4、8，我们只处理目的操作数是寄存器的情况，而不考虑目的操作数是内存的情况。

使用一个操作码表可以方便我们在代码上的处理。

```

1 static inti_2opcode

2 [[2][4]={
    //      8位操作数
    |      32位操作数

3      //r,r r,rm|rm,r r,imm| r,r r,rm|rm,r r,imm
4      {{0x8a,0x8a,0x88,0xb0},{0x8b,0x8b,0x89,0xb8}}, //mov
5      {{0x3a,0x3a,0x38,0x80},{0x3b,0x3b,0x39,0x81}}, //cmp
6      {{0x2a,0x2a,0x28,0x80},{0x2b,0x2b,0x29,0x81}}, //sub
7      {{0x02,0x02,0x00,0x80},{0x03,0x03,0x01,0x81}}, //add
8      {{0x22,0x22,0x20,0x80},{0x23,0x23,0x21,0x81}}, //and
9      {{0x0a,0x0a,0x08,0x80},{0x0b,0x0b,0x09,0x81}}, //or
10     {{0x00,0x00,0x00,0x00},{0x8d,0x8d,0x00,0x00}} //lea
11 };
12
13 int Generator::getOpCode

(Tag tag, op_type des_t,
14     op_type src_t, int len) {
15 {
16     int index = 0;
17     switch (src_t) {
18         case IMMEDIATE: index = 3; break;
19         case REGISTER: index = 2*(des_t!=REGISTER; break;
20         case MEMORY: index = 1; break;
21     }
22     return i_2opcode[tag-I_MOV][len!=8][index];
23 }

```

1) 第1~11行定义的数组*i_2opcode*记录了我们需要处理的双操作数指令操作数组合对应的操作码。数组的每一行表示一种指令对应的操作码，每一行内的第一个子数组表示8位操作数指令的操作码，第二个子数组表示32位操作数的操作码。每个子数组都是一个四元组，分别表示双寄存器操作数、寄存器-内存操作数、内存-寄存器操作数、寄存器-立即数操作数对应的操作码。

2) 函数*getOpCode*根据操作码和操作数类型以及长度计算操作码的二进制表示。对比汇编器*Tag*内对标签定义的顺序不难理解操作码的计算方式。

对双操作数指令的处理流程为:

```
1 void Generator::gen2op
(Tag tag, op_type des_t,
2   op_type src_t, int len) {
3   int opcode=getOpCode
   (tag, des_t, src_t, len);
4   switch(modrm.mod) {
5     case -1: {
//reg32 <- imm32
6       if(tag == I_MOV) {
//mov
7         opcode += modrm.reg;
8       } else {
//cmp sub add and or
9         int reg_codes[]={7, 5, 0, 4, 1}; //操作码补充字段

10        modrm.mod=3;
11        modrm.rm=modrm.reg;
12        modrm.reg=reg_codes[tag-I_CMP];
13      }
14
15      writeBytes(opcode,1); //操作码

16      if(tag!=I_MOV) writeModRM(); //modrm
17      processRel
(R_386_32); //重定位

18      writeBytes(instr.imm32,len); //立即数

19      break;
20    }
21    case 0:
//[reg32] <->reg
22      writeBytes(opcode,1);
23      writeModRM();
24      if(modrm.rm==5) {
//[disp32]
25        processRel
(R_386_32); //重定位

26      instr.writeDisp();
27    }else if(modrm.rm==4) {
28      writeSIB();
//sib
29    }
30    break;
31    case 1:
//[reg32+disp8] <->reg
32      writeBytes(opcode,1);
33      writeModRM();
34      if(modrm.rm==4) writeSIB();
35      instr.writeDisp();
```

```

36         break;
37     case 2:
38         // [reg32+disp32] <-> reg
39         writeBytes(opcode, 1);
40         writeModRM();
41         if(modrm.rm==4) writeSIB();
42         instr.writeDisp();
43         break;
44     case 3:
45         // reg <-> reg
46         writeBytes(opcode, 1);
47         writeModRM();
48         break;
49     }
50 }

```

- 1) 第3行首先调用getOpCode获取指令操作码的二进制表示opcode。
- 2) 第4~47行根据modrm的mod字段对不同的操作数类型组合进行处理。其中mod为-1时表示指令中有立即数操作数，其他情况分别对应x86指令的ModR/M的mod字段的本身含义（参考第5章对ModR/M字段的描述）。
- 3) 第5~20行处理源操作数是32位立即数（为了简化起见，将8位立即数扩展为32位）、目的操作数是32位寄存器的情况。除了lea指令，其他双操作数指令都有可能使用32位立即数。其中，mov指令的操作码定义为“b8+reg”，即0xb8是mov指令的组属性操作码，reg保存了目的寄存器的编号。在递归下降子程序operand对寄存器操作数的处理中，将目的寄存器编号保存到modrm.reg字段，源寄存器编号保存到modrm.rm字段（如果有的话），因此mov指令的操作码应该是0xb8+modrm.reg的值。虽然我们使用modrm.reg字段保存了目的寄存器

的编号，但是mov指令内并不包含ModR/M字段，这里仅仅是使用它保存寄存器的编号而已。

对于其他指令，操作码定义为“81/reg”，即0x81是指令的组属性操作码，reg即modrm.reg字段。对于指令cmp、sub、add、and、or，对应的操作码定义分别为“81/7”、“81/5”、“81/0”、“81/4”、“81/1”。这些指令是包含ModR/M字段的，其中modrm.mod=3、modrm.reg保存了以上操作码的补充码、modrm.rm字段保存了目的寄存器的编号（原来的modrm.reg字段）。

最后，调用writeBytes输出操作码opcode，并除了mov指令外，调用writeModRM输出modrm字段。在输出立即数instr.imm32之前，我们需要调用processRel处理可能存在的重定位项。这是因为每次输出操作时，我们都会自动累加当前段的逻辑地址lb_record: : curAddr，重定位项内需要记录该地址作为重定位位置的段内偏移。在输出立即数之前处理可能的重定位项保证了重定位位置恰好是立即数在段内的偏移地址。

4) 第21~30行处理了内存操作数是寄存器间址的情况。这里首先直接输出操作码opcode和modrm字段。然后处理modrm.rm字段为5的情况，它表示内存操作数使用32位偏移直接寻址。类似地，在调用writeDisp输出偏移字段之前，仍需要调用processRel处理可能的重定位

项。最后处理modrm.rm字段为4的情况，它表示指令包含sib字段，因此调用writeSIB输出sib字段。

5) 第31~36行处理内存操作数是8位基址寻址的情况。这里仍是直接输出操作码opcode和modrm字段，然后处理modrm.rm字段为4时输出sib字段，最后输出偏移字段。此处不需要重定位项的处理，因为我们的编译器生成的基址寻址指令的偏移都是数字常量，不会产生重定位。

6) 第37~42行处理内存操作数是32位基址寻址的情况。这里的处理流程与8位基址寻址完全相同，函数writeDisp会自动根据偏移字段的长度正确输出。

7) 第43~46行处理双寄存器操作数的情况。这种情况最简单，直接输出操作码opcode和modrm字段即可。

这里，我们列出输出函数writeBytes、writeModRM、writeSIB和writeDisp的定义。

```
1  /*
2      按照小端顺序 (
    little endian) 输出指定长度数据

3      len=1: 输出第
4  4字节

4      len=2: 输出第
3, 4字节
```

```

5         len=4: 输出第
1,2,3,4字节

6  */
7  void Generator::writeBytes
(int value,int len) {
8         lb_record::curAddr+=len;           //计算
地址

9         if(scanLop==2) {
10             fwrite(&value,len,1,fout);
11         }
12 }
13
14 void Generator::writeModRM
() {
15     if(modrm.mod!=-1) {
16         int byte = (modrm.mod << 6) +
17             (modrm.reg << 3) + modrm.rm;
18         writeBytes(byte,1);
19     }
20 }
21 }
22
23 void Generator::writeSIB
() {
24     if(sib.scale!=-1) {
25         int byte = (sib.scale << 6) +
26             (sib.index << 3) + sib.base;
27         writeBytes(byte,1);
28     }
29 }
30 }
31
32 void Inst::writeDisp
() {
33     if(displLen) {
34         generator.writeBytes(displ,displLen);
35         displLen=0;
36     }
37 }

```

1) 函数writeBytes在汇编器的第二遍扫描时，根据参数len的长度将value按照小字节序写入输出流fout指向的临时文件中。无论汇编器是第几遍扫描，writeBytes都会正常累加lb_record: : curAddr的值，这样就可以保证每次扫描都能处理到正确的段内偏移以及段大小。

2) 函数writeModRM和writeSIB分别将modrm和sib对象拼接为单字节，并调用writeBytes进行输出。

3) 函数writeDisp根据偏移字段的长度dispLen，调用writeBytes将偏移字段disp正确输出。

6.5.2 单操作数指令

相比于双操作数指令，单操作数指令的处理不具有一般性。需要处理的单操作数指令有call、int、imul、idiv、neg、inc、dec、jmp、je、jne、sete、setne、setg、setge、setl、setle、push和pop共18种指令。类似地，我们构造一个简单的单操作数指令的操作码表。

```
1 static int i_1opcode
[]={
2     //call, int, imul, idiv, neg, inc, dec, jmp
3     0xe8, 0xcd, 0xf7, 0xf7, 0xf7, 0x40, 0x48, 0xe9,
4     //je, jne
5     0x84, 0x85,
6     //sete, setne, setg, setge, setl, setle
7     0x94, 0x95, 0x9f, 0x9d, 0x9c, 0x9e,
8     //push pop
9     0x50, 0x58
10 };
```

需要说明的是，指令je、jne、sete、setne、setg、setge、setl、setle的操作码实际为双字节操作码，这里省略了转移操作码0x0f。对于其他指令，根据操作数的不同，操作码也会发生变化，这里只列举了某一种操作数情况下的操作码，具体的操作码的值在处理时会详细说明。

针对单字节指令的处理如下。

```
1 void Generator::gen1op
(Tag tag, int opr_t, int len) {
2     int opcode = i_1opcode[tag-I_CALL];
3     if(tag==I_CALL||tag>=I_JMP&&tag<=I_JNE) {
```

```

4          if(tag!=I_CALL&&tag!=I_JMP) {
5              writeBytes(0x0f, 1);
6          // 转义操作码
7      }
8          writeBytes(opcode);
9          // 操作码
10         int addr = processRel
11         (R_386_PC32) ?                // 重定位
12         lb_record::curAddr : instr.imm32;
13         int pc = lb_record::curAddr+4;                // pc地
14         址
15
16         writeBytes(addr-pc, 4);
17         // 相对地址
18
19     } else if (tag>=I_SETE&&tag<=I_SETLE) {
20         modrm.mod=3;
21         modrm.rm=modrm.reg;
22         modrm.reg=0;
23         writeBytes(0x0f, 1);
24         // 转义操作码
25
26         writeBytes(opcode,1);
27         // 操作码
28
29         writeModRM();
30         //modrm
31
32     } else if (tag==I_INT) {
33         writeBytes(opcode,1);
34         // 操作码
35
36         writeBytes(instr.imm32,1);
37         // 立即数
38
39     } else if (tag==I_PUSH) {
40         if(opr_t==IMMEDIATE) opcode=0x68;
41         else opcode += modrm.reg;
42         writeBytes(opcode,1);
43         // 操作码
44
45         if(opr_t==IMMEDIATE)
46             writeBytes(instr.imm32,4);
47         // 立即数
48
49     } else if(tag==I_INC||tag==I_DEC) {
50         if(len==1){
51             //reg8
52             opcode=0xfe;
53         }
54     }

```

```

34                                     int reg_codes[]={0, 1};
//操作码补充字段

35                                     modrm.mod=3;
36                                     modrm.rm=modrm.reg;
37                                     modrm.reg=reg_codes[tag-I_INC];
38                                 } else {
//reg32
39                                     opcode += modrm.reg;
40                                 }
41
42                                     writeBytes(opcode, 1);
//操作码

43                                     if(len==1) writeModRM();
//modrm
44                                 } else if(tag==I_NEG) {
45                                     if(len==1) opcode=0xf6;
46                                     modrm.mod=3;
47                                     modrm.rm=modrm.reg;
48                                     modrm.reg=3;
49
50                                     writeBytes(opcode,1);
//操作码

51                                     writeModRM();
//modrm
52                                 } else if(tag==I_POP) {
53                                     opcode+=modrm.reg;
54                                     writeBytes(opcode,1);
//操作码

55                                 } else if(tag==I_IMUL||tag==I_IDIV) {
56                                     int reg_codes[]={5, 7};
//操作码补充字段

57                                     modrm.mod=3;
58                                     modrm.rm=modrm.reg;
59                                     modrm.reg=reg_codes[tag-I_IMUL];
60
61                                     writeBytes(opcode,1);
//操作码

62                                     writeModRM();
//modrm
63                                 }
64 }

```

1) 第3~11行处理跳转类指令的情况。之所以将跳转类指令（使用相对地址的指令）统一处理，因为它们涉及重定位项的处理。我们需要处理的跳转类指令包含call、jmp、je、jne共4种指令。对于Jcc指

令，在输出操作码opcode之前，需要先输出转义操作码0x0f。接下来便是输出相对地址，而相对地址的值与指令是否需要重定位是相关的。首先调用processRel处理可能存在的相对地址重定位，如果重定位成功，则说明立即数字段instr.imm32保存的目标地址无效，那么我们将目标地址设置为当前段内偏移lb_record: : curAddr，否则正常使用instr.imm32作为目标地址。变量pc保存了下一条指令的地址，即lb_record: : curAddr+4，最终目标地址addr与pc的差值就是指令中相对地址的值。

我们发现，按照上述处理的流程，当发生相对地址重定位时，指令中的相对地址为常量值-4。而未发生相对地址重定位时，指令中的相对地址仍是原本的相对地址。之所以这样繁琐地处理是为了与链接器重定位操作保持一致（参考第7章关于重定位操作的描述），在重定位时会读取指令中保存的相对地址的值，然后根据该值计算最终链接后的相对地址。

2) 第12~20行处理SETcc类指令。这类指令也是双字节操作码，且用modrm.reg字段补充操作码，操作码定义为“SETcc/0”。其中modrm.mod字段为3、modrm.reg字段为0、modrm.rm字段为寄存器操作数的编号。输出指令时，首先输出转义操作码，然后输出opcode和modrm字段。

3) 第21~23行处理int指令。该指令是单字节操作码指令，操作数是8位立即数。因此直接输出opcode和instr.imm32低8位即可。

4) 第24~30行处理push指令。我们使用的push指令的操作数有两类：32位立即数和32位寄存器。当操作数为立即数时，push指令的操作码是0x68，操作码后紧跟32位立即数。当操作数为寄存器时，push指令操作码为组属性操作码，使用寄存器编号进行补充，操作码定义为“50+reg”。单操作数指令操作码表保存的push指令操作码正是此值，因此输出操作码前需要将opcode累加操作数寄存器的编码值。

5) 第31~43行处理inc和dec指令。我们只使用了指令操作数为寄存器的情况，不过该类指令的操作码定义与操作数长度相关。当操作数为32位寄存器时，操作码定义为“inc/dec+reg”，即使用操作数寄存器编号补充操作码，组属性操作码的值见操作码表。当操作数为8位寄存器时，inc指令的操作码定义为“fe/0”，dec指令的操作码定义为“fe/1”，即使用modrm.reg补充操作码，因此需要输出0xfe和modrm字段。

6) 第44~52行处理neg指令。neg指令操作码定义也与操作数长度相关。当操作数长度为32位寄存器时，操作码定义为“f7/3”。当操作数长度为8位寄存器时，操作码定义为“f6/3”。因此输出组属性操作码后，还要输出modrm字段。

7) 第52~54行处理pop指令。pop指令操作数为32位寄存器时，操作码定义为“58+reg”，即使用寄存器操作数编码补充操作码。

8) 第55~62行处理imul和idiv指令。这两个指令使用32位寄存器操作数时，使用相同的组属性操作码，操作码定义分别为“f7/5”和“f7/7”。因此需要输出0xf7和modrm字段。

6.5.3 零操作数指令

最后讨论零操作数指令，该类指令我们只处理一种指令`ret`，指令二进制编码为`0xc3`。

```
1 static int i_opcode
   []=
2 {
3     //ret
4     0xc3
5 };
6
7 void Generator::gen0op
   (Tag tag) {
8     int opcode=i_opcode[tag-I_RET];
9     writeBytes(opcode,1);           //操作码
10 }
```

6.6 目标文件生成

汇编器两遍扫描结束后，段表信息、符号表信息和重定位表信息已经保存到ELF文件对象中，代码段的内容被放入一个临时文件内，而数据段的内容可以直接从汇编器的符号表中取出包含数据的符号输出即可。最终汇编器会根据图6-6描述的ELF文件结构，将ELF文件头、代码段、数据段、段表字符串表、段表、符号表、字符串表、重定位表的内容按序输出，生成合法的可重定位目标文件。

整个过程分为两大阶段。

1) **ELF文件结构组装**。根据扫描得到ELF文件信息，完善ELF文件结构的数据。包括文件头各个字段的值、串表的内容以及引用串表内的字符串偏移信息（如段表项的段名字段`sh_name`、符号表项的符号名字段`st_name`）、重定位表项等。

2) **ELF文件结构输出**。输出文件头、代码段、数据段、段表字符串表、段表、符号表、字符串表、重定位表的内容。任何两个文件结构因为对齐需要产生了空隙，则使用0填充补齐。

首先看ELF文件结构的组装。

```

1 void Elf_file::assemObj
2     {
3         //所有段名
4         vector<string> AllSegNames = shdrNames;
5         AllSegNames.push_back(".shstrtab");
6         AllSegNames.push_back(".symtab");
7         AllSegNames.push_back(".strtab");
8         AllSegNames.push_back(".rel.text");
9         AllSegNames.push_back(".rel.data");
10        //段索引
11
12        hash_map<string,int,string_hash> shIndex
13        ;
14        //段名索引
15
16        hash_map<string,int,string_hash> shstrIndex
17        ;
18        //建立索引
19
20        for (int i=0;i<AllSegNames
21        .size();++i){
22            string name = AllSegNames[i];
23            shIndex[name] = i;
24            shstrIndex[name] = shstrtab.size();
25            shstrtab
26            += name;
27            //保存数据
28
29            shstrtab.push_back('\0');
30        }
31        //符号索引
32
33        hash_map<string,int,string_hash> symIndex
34        ;
35        //符号名索引
36
37        hash_map<string,int,string_hash> strIndex
38        ;
39        //建立索引
40
41        for (int i=0;i<symNames
42        .size();++i){
43            string name = symNames[i];
44            symIndex[name] = i;
45            strIndex[name] = strtab.size();
46            strtab

```

```

+= name; //保存数据

33         strtab.push_back('\0');
34     }
35
36         //更新符号表符号名索引

37     for (int i=0;i<symNames
.size();++i){
38         string name = symNames[i];
39         symTab
[name]->st_name=strIndex[name];
40     }
41
42         //处理重定位表

43     for(int i=0;i<relTab
.size();i++){
44         Elf32_Rel*rel=new Elf32_Rel();
45         rel->r_offset=relTab[i]->r_offset; //重定位位置

46         rel->r_info=ELF32_R_INFO(
47             symIndex[relTab[i]->r_name], //重定
位符号

48             ELF32_R_TrPE(relTab[i]->r_info) //重定位
类型

49
50         if(relTab[i]->SegName==".text") //重
定位段

51             relTextTab
.push_back(rel);
52         else if(relTab[i]->SegName==".data")
53             relDataTab
.push_back(rel);
54         else
55             delete rel;
56     }
57
58         //处理文件头

59     char magic[] = {
//魔数

60         0x71, 0x45, 0x4c, 0x46,
61         0x01, 0x01, 0x01, 0x00,
62         0x00, 0x00, 0x00, 0x00,
63         0x00, 0x00, 0x00, 0x00
64     };
65

```

```

66     memcpy(&ehdr.e_ident, magic, sizeof(magic));
67     ehdr.e_type=ET_REL;
68     ehdr.e_machine=EM_386;
69     ehdr.e_version=EV_CURRENT;
70     ehdr.e_entry=0;
71     ehdr.e_phoff=0;
72     ehdr.e_shoff=0;
73     ehdr.e_flags=0;
74     ehdr.e_ehsize=sizeof(Elf32_Ehdr);
75     ehdr.e_phentsize=0;
76     ehdr.e_phnum=0;
77     ehdr.e_shentsize=sizeof(Elf32_Shdr);
78     ehdr.e_shnum=AllSegNames.size();
79     ehdr.e_shstrndx=shIndex[".shstrtab"];
80
81     int curOff = sizeof(ehdr
);
//文件头, 已对齐

82
83     curOff += dataLen;
//已有段, 已对齐

84
85     //添加新的段表项

86     addShdr(".shstrtab
", SHT_STRTAB, 0, 0, curOff,
87         shstrtab.size(), SHN_UNDEF, 0, 1, 0);
88     curOff += shstrtab.size();
89     curOff += (4-curOff%4)%4;
//对齐

90
91     ehdr.e_shoff
= curOff;
//段表偏移

92     curOff += ehdr.e_shnum*ehdr.e_shentsize;
//段表, 已对齐

93
94     addShdr(".symtab
", SHT_SYMTAB, 0, 0, curOff,
95         symNames.size()*sizeof(Elf32_Sym),
96         shIndex[".strtab"], 0, 1, sizeof(Elf32_Sym));
97     curOff += symNames.size()*sizeof(Elf32_Sym);
//已对齐

98
99     addShdr(".strtab
", SHT_STRTAB, 0, 0, curOff,
100         strtab.size(), SHN_UNDEF, 0, 1, 0);
101     curOff += strtab.size();
102     curOff += (4-curOff%4)%4;
//对齐

```

```

103
104         addShdr(".rel.text
", SHT_REL, 0, 0, curOff,
105             relTextTab.size()*sizeof(Elf32_Rel),
106             shIndex[".symtab"], shIndex[".text"], 1,
107             sizeof(Elf32_Rel));
108         curOff += relTextTab.size()*sizeof(Elf32_Rel);           //已对齐

109
110         addShdr(".rel.data
", SHT_REL, 0, 0, curOff,
111             relDataTab.size()*sizeof(Elf32_Rel),
112             shIndex[".symtab"], shIndex[".data"], 1,
113             sizeof(Elf32_Rel));
114         curOff += relDataTab.size()*sizeof(Elf32_Rel);           //已对齐

115
116         //更新段表段名索引

117         for (int i=0;i<AllSegNames.size();++i){
118             string name = AllSegNames[i];
119             shdrTab
[name]->sh_name=shstrIndex[name];
120         }
121     }

```

1) 第2~8行，我们使用AllSegNames记录所有的段名，包括shdrNames记录的段（空段表项（初始化时加入）、汇编代码中扫描得到的代码段“.text”和数据段“.data”），以及ELF文件内部使用的段：“.shstrtab”、“.symtab”、“.strtab”、“.rel.text”和“.rel.data”。

2) 第10~21行建立段索引shIndex和段名索引shstrIndex。shIndex记录了每个段表项在AllSegNames的索引位置，段表会根据AllSegNames记录的段名顺序生成各个段表项。shstrIndex记录了每个段名在段表字符串表“.shstrtab”内的位置，我们按AllSegNames的顺序拼接所有的段名形成段表字符串表，拼接时注意使用‘\0’分割不同的段名。

3) 第23~34行建立符号索引`symIndex`和符号名索引`strIndex`。
`symIndex`记录了每个符号表项（包括初始化时添加的空符号表项）在符号名列表`symNames`的索引位置，符号表会根据`symNames`记录的符号名顺序生成各个符号表项。`strIndex`记录了每个符号名在字符串表“`.strtab`”内的位置，我们按`symNames`的顺序拼接所有的符号名形成字符串表，拼接时注意使用‘\0’分割不同的符号名。

4) 第36~40行扫描符号表`strTab`，然后将每个符号表项的`st_name`字段设为符号名在字符串表内的索引，完成符号表信息的补充。

5) 第42~56行处理ELF文件对象内记录的重定位表项信息。首先生成重定位表项`Elf32_Rel`，设置重定位位置`r_offset`，使用`ELF32_R_INFO`宏设置重定位符号（符号索引）和重定位类型`r_info`。最后根据重定位的段名将重定位信息分为两个部分：代码段重定位表`relTextTab`和数据段重定位表`relDataTab`。

6) 第58~79行处理ELF文件头，按照第5章描述的ELF文件头的内容设置对应字段。其中，`e_type`设为`ET_REL`表示文件类型是重定位目标文件，`e_ehsize`设为`sizeof (Elf32_Ehdr)`表示文件头大小，`e_shentsize`设为`sizeof (Elf32_Shdr)`表示段表项的大小，`e_shnum`设为`AllSegNames`的大小表示段表项个数，`e_shstrndx`设为“`.shstrtab`”的段索引。另外，`e_shoff`暂时初始化为0，因为还未确定段表的文件偏移。

7) 第81~83行将curOff初始化为ELF文件头的大小，然后累加dataLen大小。dataLen记录了汇编器扫描的所有段（代码段和数据段）对齐后的大小。因为最终curOff为文件头、代码段、数据段的总大小，即“.shstrtab”段的文件偏移。

8) 第85~89行添加“.shstrtab”的段表项到段表shdrTab。其中比较关键的字段有段名“.shstrtab”、段类型SHT_STRTAB、段文件偏移curOff、段大小shstrtab.size（）、对齐大小1（即该段的文件偏移不进行对齐）等。然后将curOff累加当前段大小、对齐（后续文件结构要求的对齐方式），继续处理下一个文件结构。

9) 第91~92行处理段表的信息。首先将文件头的e_shoff设为curOff，即段表的偏移。然后将curOff累加段表的大小：段表项个数*段表项大小。

10) 第94~97行添加“.symtab”的段表项到段表shdrTab。其中比较关键的字段有段名“.symtab”、段类型SHT_SYMTAB、段文件偏移curOff、段大小（符号表项个数*符号表项大小）、sh_link记录符号表使用的串表“.strtab”的段索引、sh_info记录第一个全局符号的符号索引（由于我们没有在符号表内区分全局符号的区域，因此设为0，这样链接器会扫描所有的符号，根据符号的全局属性进行符号解析）、对齐大小1（即该段的文件偏移不进行对齐）、符号表项大小sizeof

(Elf32_Sym) 等。然后将curOff累加当前段大小，继续处理下一个文件结构。

11) 第99~102行添加“.strtab”的段表项到段表shdrTab。其中比较关键的字段有段名“.strtab”、段类型SHT_STRTAB、段文件偏移curOff、段大小strtab.size ()、对齐大小1（即该段的文件偏移不进行对齐）等。然后将curOff累加当前段大小、对齐（后续文件结构要求的对齐方式），继续处理下一个文件结构。

12) 第104~108行添加“.rel.text”的段表项到段表shdrTab。其中比较关键的字段有段名“.rel.text”、段类型SHT_REL、段文件偏移curOff、段大小（代码段重定位表项个数*重定位表项个数）、对齐大小1（即该段的文件偏移不进行对齐）、重定位表项大小sizeof (Elf32_Rel) 等。然后将curOff累加当前段大小，继续处理下一个文件结构。

13) 第110~114行添加“.rel.data”的段表项到段表shdrTab。其中比较关键的字段有段名“.rel.data”、段类型SHT_REL、段文件偏移curOff、段大小（数据段重定位表项个数*重定位表项个数）、对齐大小1（即该段的文件偏移不进行对齐）、重定位表项大小sizeof (Elf32_Rel) 等。然后将curOff累加当前段大小，继续处理下一个文件结构。

14) 第116~120行扫描段表shdrTab，然后将每个段表项的sh_name
字段设为段名在段表字符串表内的索引，完成段表信息的补充。

到这里，已经完成了ELF目标文件信息的组装。

最后，便是ELF文件结构的输出。

```
1 void Elf_file::writeElf
2
3 {
4     int padNum = 0;
5     char pad[1]={0};
6
7     //文件头
8
9     fwrite(&ehdr
10    ,ehdr.e_ehsize,1,fout);
11
12     // .text
13     char buffer[1024]={0};
14     int count = -1;
15     while(count) {
16         count = fread(buffer,1,1024,fin
17     );
18     fwrite(buffer,1,count,fout);
19 }
20
21 // .data
22 padNum = shdrTab[".data"]->sh_offset
23         - shdrTab[".text"]->sh_offset
24         - shdrTab[".text"]->sh_size;
25 fwrite(pad,sizeof(pad),padNum,fout);
26 table.write
27
28 ();
29
30 // .shstrtab
31 padNum = shdrTab[".shstrtab"]->sh_offset
32         - shdrTab[".data"]->sh_offset
33         - shdrTab[".data"]->sh_size;
34 fwrite(pad,sizeof(pad),padNum,fout);
35 fwrite(shstrtab
36 .c_str(),shstrtab.size(),1,fout);
37
38 //段表
39
40 padNum = ehdr.e_shoff
41         - shdrTab[".shstrtab"]->sh_offset
42         - shdrTab[".shstrtab"]->sh_size;
```

```

34     fwrite(pad, sizeof(pad), padNum, fout);
35     for(int i=0; i<shdrNames.size(); ++i){
36         Elf32_Shdr* sh=shdrTab
[shdrNames[i]];
37         fwrite(sh, ehdr.e_shentsize, 1, fout);
38     }
39
40     //符号表

41     for(int i=0; i<symNames.size(); ++i){
42         Elf32_Sym* sym=symTab
[symNames[i]];
43         fwrite(sym, sizeof(Elf32_Sym), 1, fout);
44     }
45
46     //.strtab
47     fwrite(strtab
.c_str(), strtab.size(), 1, fout);
48
49     //.rel.text
50     padNum = shdrTab[".rel.text"]->sh_offset
51             - shdrTab[".strtab"]->sh_offset
52             - shdrTab[".strtab"]->sh_size;
53     fwrite(pad, sizeof(pad), padNum, fout);
54     for(int i=0; i<relTextTab.size(); ++i){
55         Elf32_Rel* rel=relTextTab
[i];
56         fwrite(rel, sizeof(Elf32_Rel), 1, fout);
57     }
58
59     //.rel.data
60     for(int i=0; i<relDataTab.size(); ++i){
61         Elf32_Rel* rel=relDataTab
[i];
62         fwrite(rel, sizeof(Elf32_Rel), 1, fout);
63     }
64 }

```

1) 首先输出ELF文件头ehdr，调用fwrite将该结构输出到文件即可。

2) 第8~14行输出代码段的二进制内容。其中，输入流文件指针fin指向指令生成时产生的代码段临时文件，这里只需要将临时文件的内容输出到目标文件即可。

3) 第16~21行输出数据段的二进制内容。首先使用0填充“.data”段和“.text”段因为对齐产生的间隙。然后调用符号表table的write函数输出数据段的内容。

4) 第23~28行输出“.shstrtab”段。首先使用0填充“.shstrtab”段和“.data”段因为对齐产生的间隙。然后输出shstrtab保存的段名字符串内容。

5) 第30~38行输出段表。首先使用0填充段表和“.shstrtab”段因为对齐产生的间隙。然后按照shdrNames保存的段名顺序输出shdrTab保存的段表项内容。

6) 第40~44行输出“.symtab”段。只需按照symNames保存的符号名顺序输出symTab保存的符号表项内容即可。

7) 第46~47行输出“.strtab”段。只需输出strtab保存的符号名字符串内容即可。

8) 第49~57行输出“.rel.text”段。首先使用0填充“.rel.text”段和“.strtab”段因为对齐产生的间隙。然后输出relTextTab保存的代码段重定位表项内容。

9) 第59~63行输出“.rel.data”段。只需输出relDataTab保存的数据段重定位表项内容即可。

在输出数据段的内容时，调用符号表的**write**函数实现。

```
1 void lb_record::write
   () {
2     for(int i=0; i<times; i++) {
3         list<int>::iterator j = cont.begin();
4         for(; j != cont.end(); j++) {
5             generator.writeBytes
   (*j, this->len);
6         }
7     }
8 }
9
10 void Table::write
   () {
11     for(int i=0; i<defLbs.size(); i++) {
12         defLbs
   [i]->write();
13     }
14 }
```

符号表会遍历所有的已经保存的包含数据的符号**defLbs**，然后逐个输出符号的数据内容。符号对象的**write**方法会根据符号定义的重复次数和长度，调用指令生成器的**writeBytes**方法将符号的数据按照小字节序输出。

至此，我们将可重定位目标文件的内容输出完毕，完成了汇编器的最后一步操作。使用**readelf**命令，可以查看验证我们输出的目标文件结构的正确性。

6.7 本章小结

本章根据已设计的汇编器结构，分别从词法分析、语法分析、符号表管理、表信息生成、指令生成和目标文件生成的角度描述了一个简单的汇编器实现。我们发现汇编器和编译器在实现上有很大的相似性，尤其是在词法分析和语法分析部分，有很多相似的逻辑和流程。不过汇编器也有其独特性，在生成ELF文件和处理汇编指令的翻译过程中，涉及了大量计算机底层的内容。笔者在实现汇编器和整理本章内容的时候，也会反复查阅Intel的x86指令手册以及ELF相关的资料，以保证每个指令的操作码和ELF文件结构字段描述的正确性，避免误导读者。

相信经过本章的描述，大家对汇编器的实现有了比较清晰的了解。我们目前终于弄清了一段高级语言代码是如何一步步转化为目标文件的。不过目标文件仅仅存储了代码的二进制表示，还不能在操作系统中正常执行。在接下来的第7章中，我们将介绍如何将汇编器生成的目标文件处理、组装成一个真正可以执行的文件，以验证我们自定义的高级语言代码和编译系统实现的正确性。

第7章 链接器构造

合纵缔交，相与为一。

——《史记》

在编译系统中，链接器扮演类似“胶水”的角色。它把汇编器处理生成的可重定位目标文件黏合、拼接为一个可执行的ELF文件。然而，链接器并非机械地拼接目标文件，它还需要完成汇编阶段无法完成的段地址分配、符号地址计算以及数据/指令内容修正的工作。这三个主要任务涉及了链接器工作的核心流程：地址空间分配、符号解析和重定位。

在可重定位目标文件的段表项中，段的虚拟地址都是默认设为0。这是因为在汇编器处理阶段，是不可能知道段的加载地址的。链接器的地址空间分配操作的主要目的是为段指定加载地址。

在确定了段加载地址（简称段基址）后，根据目标文件内符号的段内偏移地址，可以计算得到符号的虚拟地址（简称符号地址）。链接器的符号解析操作并不止于计算符号地址，它还需要分析目标文件之间的符号引用的情况，计算目标文件内引用的外部符号的地址。

符号解析之后，所有目标文件的符号地址都已经确定。链接器通过重定位操作，修正代码段或数据段内引用的符号地址。

最后，链接器将以上操作处理后的文件信息导出为可执行ELF文件，完成链接的工作。

参考图2-17描述的链接器的结构设计，我们在本章详细阐述链接器每个功能模块的实现。

7.1 信息收集

对链接器来说，其输入是一系列的可重定位目标文件。链接器欲完成后续的工作，必须逐个扫描目标文件，提取需要的信息进行处理。因此，我们需要建立必要的数据结构缓存链接器需要的信息。

7.1.1 目标文件信息

首先，需要建立ELF文件对象，保存扫描的目标文件信息。第6章已经构建了Elf_file对象，不过该对象的主要功能是将ELF文件结构信息写入目标文件。而链接器需要扫描ELF文件的内容，因此需要添加必要的ELF文件读取操作。正如汇编器生成目标文件时比较关心段表、符号表、重定位表信息那样，链接器扫描目标文件时也会着重关注这三个文件结构的信息。参考第6章对ELF目标文件结构的构造方式，读取ELF目标文件结构的实现代码为：

```
1 void Elf_file::readElf
  (const string dir) {
2     //打开目标文件

3     elf_dir=dir;
4     FILE*fp=fopen(elf_dir.c_str(),"rb");
5
6     //文件头

7     rewind(fp);
8     fread(&ehdr
, sizeof(Elf32_Ehdr), 1, fp);
9
10    //程序头表

11    if(ehdr.e_type==ET_EXEC) {
12        fseek(fp,ehdr.e_phoff,0);
13        for(int i=0;i<ehdr.e_phnum;++i) {
14            Elf32_Phdr*phdr=new Elf32_Phdr();
15            fread(phdr,ehdr.e_phentsize,1,fp);
16            phdrTab

        .push_back(phdr);
17        }
18    }
19
20    //.shstrtab表项
```

```

21     Elf32_Shdr shstrTab;
22     fseek(fp, ehdr.e_shoff+ehdr.e_shentsize*ehdr.e_shstrndx, 0);
23     fread(&shstrTab, ehdr.e_shentsize, 1, fp);
24
25     //.shstrtab
26     char*shstrTabData=new char[shstrTab.sh_size];
27     fseek(fp, shstrTab.sh_offset, 0);
28     fread(shstrTabData
, shstrTab.sh_size, 1, fp);
29
30     //段表

31     fseek(fp, ehdr.e_shoff, 0);
32     for(int i=0; i<ehdr.e_shnum; ++i) {
33         Elf32_Shdr*shdr=new Elf32_Shdr();
34         fread(shdr, ehdr.e_shentsize, 1, fp);
35         string name(shstrTabData+shdr->sh_name);
36         shdrNames

.push_back(name);
37         shdrTab

[name]=shdr;
38     }
39
40     //.strtab
41     Elf32_Shdr *strTab=shdrTab[".strtab"];
42     char*strTabData=new char[strTab->sh_size];
43     fseek(fp, strTab->sh_offset, 0);
44     fread(strTabData
, strTab->sh_size, 1, fp);
45
46     //.symtab
47     Elf32_Shdr *sh_symTab=shdrTab[".symtab"];
48     fseek(fp, sh_symTab->sh_offset, 0);
49     int symNum=sh_symTab->sh_size/sh_symTab->sh_entsize;
50     for(int i=0; i<symNum; ++i) {
51         Elf32_Sym*sym=new Elf32_Sym();
52         fread(sym, sh_symTab->sh_entsize, 1, fp);
53         string name(strTabData+sym->st_name);
54         symNames

.push_back(name);
55         symTab

[name]=sym;
56     }
57
58     //.rel.data .rel.text
59     for(int i=0; i<shdrNames.size(); i++) {
60         string shdrName = shdrNames[i];
61         Elf32_Shdr*shdr=shdrTab[shdrName];
62         if(shdr->sh_type==SHT_REL) {
63             fseek(fp, shdr->sh_offset, 0);
64             int relNum=shdr->sh_size/shdr->sh_entsize;
65             for(int j=0; j<relNum; ++j) {
66                 Elf32_Rel*rel=new Elf32_Rel();
67                 fread(rel, shdr->sh_entsize, 1, fp);
68                 string segName=shdrNames[shdr->sh_info];
69                 string symName=symNames[ELF32_R_SYM(rel->r_info)];
70                 relTab

```

```
.push_back(new RelItem(segName, rel, symName));
71         }
72     }
73 }
74
75     delete []shstrTabData;
76     delete []strTabData;
77     fclose(fp);
78 }
```

我们使用readElf函数读取ELF文件的信息。

- 1) 第2~4行打开目标文件，并将文件路径记录到elf_dir。
- 2) 第6~8行读取ELF文件头的信息到ehdr。
- 3) 第10~18行读取程序头表的信息。虽然目标文件没有该文件结构，但我们还是实现了这部分的逻辑。即从文件的e_phoff偏移处，读取e_phentsize个Elf32_Phdr对象，保存到phdrTab列表即可。
- 4) 第20~23行读取段表字符串表“.shstrtab”的段表项信息，根据e_shstrndx字段以及段表偏移和段表项大小，可计算得到“.shstrtab”的位置，然后取出该段表项对应的Elf32_Shdr对象shstrTab。
- 5) 第25~28行读取段表字符串表“.shstrtab”的数据内容。即根据段表项shstrTab记录的段偏移sh_offset和段大小sh_size，读取数据到字符缓冲区shstrTabData。
- 6) 第30~38行读取段表的信息。即从文件的e_shoff偏移处，读取e_shentsize个Elf32_Shdr对象，保存到shdrTab表即可。其中段名可以从

shstrTabData的sh_name位置获得，我们将段名按序保存到段名列表shdrNames，以方便段表项的按索引访问。

7) 第40~44行读取字符串表“.strtab”的数据内容。即根据段表项记录的段偏移sh_offset和段大小sh_size，将文件中的内容读取到字符缓冲区strTabData。理论上，对字符串表的访问方式应该是根据段表中扫描到的重定位表项（段类型为SHT_REL）的sh_link字段得到重定位表引用的符号表的段表索引，继而取出符号表的段表项，然后根据符号表的段表项的sh_link字段得到符号表引用的字符串表的段索引，最终得到字符串表的数据内容。不过，我们很清楚在汇编器生成的目标文件内只有唯一一个“.strtab”段，这里直接按名访问是一种简化的方式，后面对符号表的访问与此类似。

8) 第46~56行读取符号表“.symtab”的信息。即从段表项记录的sh_offset位置，读取sh_size/sh_entsize个Elf32_Shdr对象，保存到symTab。其中符号名从strTabData的st_name位置获得，我们将符号名按序保存到符号名列表symNames，以方便符号表项的按索引访问。

9) 第58~73行读取重定位表的信息。通过遍历取出段类型为SHT_REL的段表项，从段表项记录的sh_offset位置，读取sh_size/sh_entsize个Elf32_Rel对象。对于每个重定位表项rel，根据当前段的sh_info从shdrNames内取得重定位段名segName，根据重定位表项的r_info取出重定位符号在符号表内的索引，继而从symNames内取

得重定位符号名`symName`。根据以上信息构造`RelItem`对象，保存到`relTab`表即可。

7.1.2 段数据信息

通过读取ELF可重定位目标文件，可以将待链接的目标文件信息保存起来。然而，链接器处理的对象其实是目标文件内保存的二进制程序或数据，如代码段“.text”和数据段“.data”的内容。因此，需要定义相关的数据结构以保存目标文件内的二进制信息，辅助链接器的工作。为了保持描述的一致性，本书将段内保存的二进制信息统称为段数据。

```
1  //数据块

2  struct Block
3  {
4      char *data;           //块数据

5      unsigned int offset;  //块偏移

6      unsigned int size;    //块大小

7  };

8  //同类型段列表

9  struct SegList
10 {
11     unsigned int baseAddr; //基地址

12     unsigned int begin;    //对齐前偏移

13     unsigned int offset;   //对齐后偏移

14     unsigned int size;     //总大小
```



```

14      vector<Elf_file*>ownerList;                                //所有者文件

15      vector<Block*>blocks;                                       //数据块

16
17      void allocAddr(string name,unsigned int& base,
18                      unsigned int& off);
19      void relocAddr(unsigned int relAddr,
20                      unsigned char type,unsigned int symAddr);
21 };
22
23 hash_map<string,SegList*,string_hash>segLists

;      //所有合并段列表

```

1) 第1~6行定义Block类保存段数据的信息，其中data表示数据块的地址，offset表示数据块经过链接器处理后在可执行文件内的偏移，size表示数据块的大小。由于链接器在重定位阶段需要段数据内容，因此Block保存的数据块为重定位操作提供了数据载体。

2) 第9~21行定义SegList类保存同类型段的数据块和相关信息。其中baseAddr表示链接器为合并后的段分配的虚拟段基地址，begin表示合并后的段在对齐之前的文件偏移，offset表示对齐后的文件偏移，size表示合并后的段大小，ownerList表示包含该类型段的目标文件列表，blocks表示所有的当前段类型的段数据列表。函数allocAddr用于段的地址空间分配操作，relocAddr用于重定位操作。SegList对象保存的信息方便了链接器的段地址空间分配工作。

3) 第23行定义的segLists对象保存了所有类型的SegList对象。由于在汇编器阶段仅生成了两种类型的段：“.text”段和“.data”段，因此在简

化的链接器的实现中，`segLists`对象其实只有两个元素。

为了方便对段数据信息相关数据结构的理解，我们使用一个简单的例子说明。

如图7-1所示，链接器会将同名的段合并，如“.text”段和“.data”段。深色部分表示目标文件内段的二进制数据，`Block`对象会记录该数据的内容、大小以及合并后的偏移。比如目标文件a.o和b.o的“.text”段会被合并为可执行文件的“.text”段。合并后的段内包含原始的段内二进制数据和因为段对齐产生的段内填充数据。段内对齐会根据目标文件内保存的段对齐字段`sh_align`进行，如文件b.o的“.text”的偏移会根据4字节对齐。合并后的段也需要根据可执行文件对段的对齐要求进行对齐，如最终的“.text”段的偏移会按照链接器要求的16字节进行对齐（其他类型的段，如数据段仍是按照4字节对齐）。对齐前的文件偏移保存在`SegList::begin`内，以方便可执行文件生成时对段间的间隙进行填充。

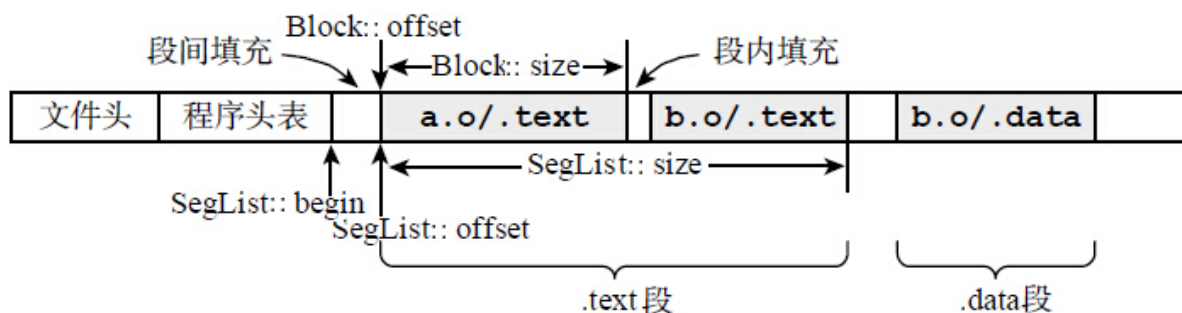


图7-1 段数据组织

由于同类型的段数据被保存在SegList的对象中，因此当需要对段的数据内容进行重定位（修改数据）操作时，只需要根据重定位的偏移地址找到对应的数据块，然后修改对应的数据即可。

7.1.3 符号引用信息

除了对目标文件的段数据进行重新组织，链接器还需要分析目标文件内符号的引用情况。之所以要分析符号的引用信息，是因为在链接器处理的目标文件中，存在未定义的符号，即对其他目标文件符号的引用。为了方便链接器符号解析的处理，一般会定义两个符号集合：一个是导出符号集合，表示所有目标文件内定义的可以被其他目标引用的全局符号集合；另一个是导入符号集合，表示所有目标文件未定义却引用其他目标文件的符号集合。

```
1  //符号引用对象

2  struct SymLink {
3      string name;                                //符号名

4      Elf_file*recv;                              //引用符号文件

5      Elf_file*prov;                              //提供符号的文件

6  };
7
8  vector<SymLink*>symLinks
9
10 ;                                                //所有符号引用信息

11
12 vector<SymLink*>symDef
13
14 ;                                                //所有符号定义信息
```

1) 第1~6行定义了SymLink对象记录符号引用的信息，其中name为符号名，recv为引用符号的目标文件，prov为定义符号的目标文件。

2) 第8~9行的symLinks表示导入符号的集合，symDef表示导出符号集合。

如图7-2所示，在目标文件a.o内定义了全局符号var和main，在目标文件b.o内定义了全局符号ext和fun，这些符号所在的段已经在符号名前标明，链接器会将这四个全局符号放入导出符号集合。另外，目标文件a.o的符号表内还保存了未定义符号ext和fun，目标文件b.o的符号表内也保存了未定义符号var，链接器将这三个未定义符号放入导入符号集合。经过链接器的符号解析处理后，会为每个导入符号找到定义它的目标文件。例如，a.o中ext和fun符号的定义在b.o文件内，以及b.o中var符号的定义在a.o文件内。

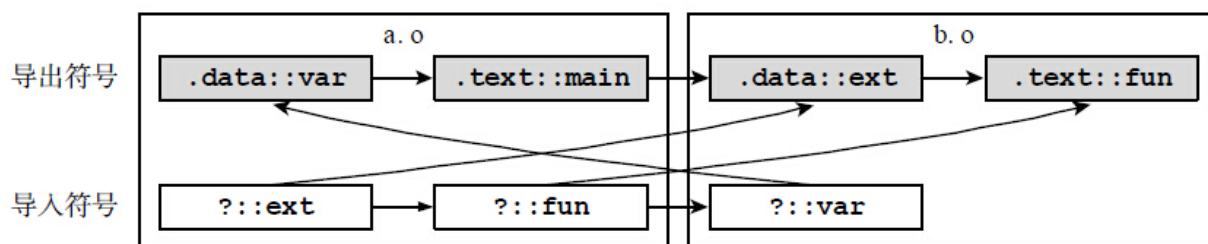


图7-2 符号引用

7.2 地址空间分配

汇编器生成目标文件时，由于无法确定段的加载地址，因此默认将段基址设置为0。链接器的第一步工作便是确定需要加载段的段基址，为待加载段指定段基址的过程称为地址空间分配。

链接器为段指定基址，需要从三个方面进行考虑。

1) 段加载的起始地址。该地址是所有加载段的起始位置，在32位Linux系统中，一般设置为0x08048000。

2) 段的拼接顺序。链接器按序扫描目标文件内同名的段，并将段的二进制数据依次“摆放”。在我们实现的链接器中，只需要按照代码段“.text”、数据段“.data”的顺序，依次处理每个目标文件内该类型的段即可。

3) 段对齐方式。段对齐包含两个层面：段文件偏移的对齐和段基址的对齐。在可重定位的目标文件内，一般将段的文件偏移对齐设置为4字节，不考虑段基址的对齐（段基址都是0，没有对齐的意义）。而在可执行文件内，会将代码段“.text”的文件偏移对齐设置为16字节，其他段的文件偏移对齐方式仍默认为4字节。而段基址的对齐则比较复杂，[需要保证](#)段的线性地址与段对应文件偏移相对于段对齐值（即[页](#)

面大小，Linux下默认为4096字节）取模相等。此处可参考第5章关于程序头表内段对齐字段p_align的解释。

图7-3给出了一个地址空间分配的例子。目标文件a.o的代码段大小为0x4a字节，数据段大小为0x08字节，b.o的代码段大小为0x21字节，数据段大小为0x04字节。

首先确定段的文件偏移。根据前面的描述，链接器会将a.o的代码段、b.o的代码段、a.o的数据段、b.o的数据段依次“摆放”。基于该顺序，可以确定每个段的文件偏移。在可执行文件的代码段之前，还有文件头和程序头表结构。其中文件头占用52字节，程序头表包含两个表项（分别用于加载代码段和数据段），占用 $2 \times \text{sizeof}(\text{Elf32_Phdr}) = 2 \times 32 = 64$ 字节，因此代码段的文件偏移为 $52 + 64 = 116$ 字节（0x74）。考虑到可执行文件代码段的文件偏移需要按16字节对齐，因此最终确定的代码段文件偏移为0x80。基于此，确定a.o的代码段文件偏移为0x80。b.o的代码段文件偏移为 $0x80 + 0x4a = 0xca$ ，按照4字节（目标文件段表内段表项的sh_align=4）对齐后为0xcc。依此类推，得到a.o的数据段文件偏移为0xf0，b.o的数据段文件偏移为0xf8。

接着，根据段的文件偏移确定段基址。从默认的段加载起始地址0x08048000开始，计算代码段和数据段的基址。将起始地址按照页大小4096字节（0x1000）对齐为0x08048000，然后累加代码段文件偏移相对于页大小的模值 $0x80 \% 0x1000 = 0x80$ ，得到代码段的最终基址为

0x08048080。基于此，得到a.o的代码段的基址为0x08048080，b.o的代码段的基址为0x080480cc，代码段结束位置的虚拟地址为0x080480ed。接下来处理数据段，将代码段的结束位置的虚拟地址按照页大小对齐后为0x08049000，累加数据段文件偏移相对于页大小的模值0xf0%0x1000=0xf0，得到数据段的最终基址为0x080490f0。基于此，得到a.o的数据段基址为0x080490f0，b.o的数据段基址为0x080490f8。

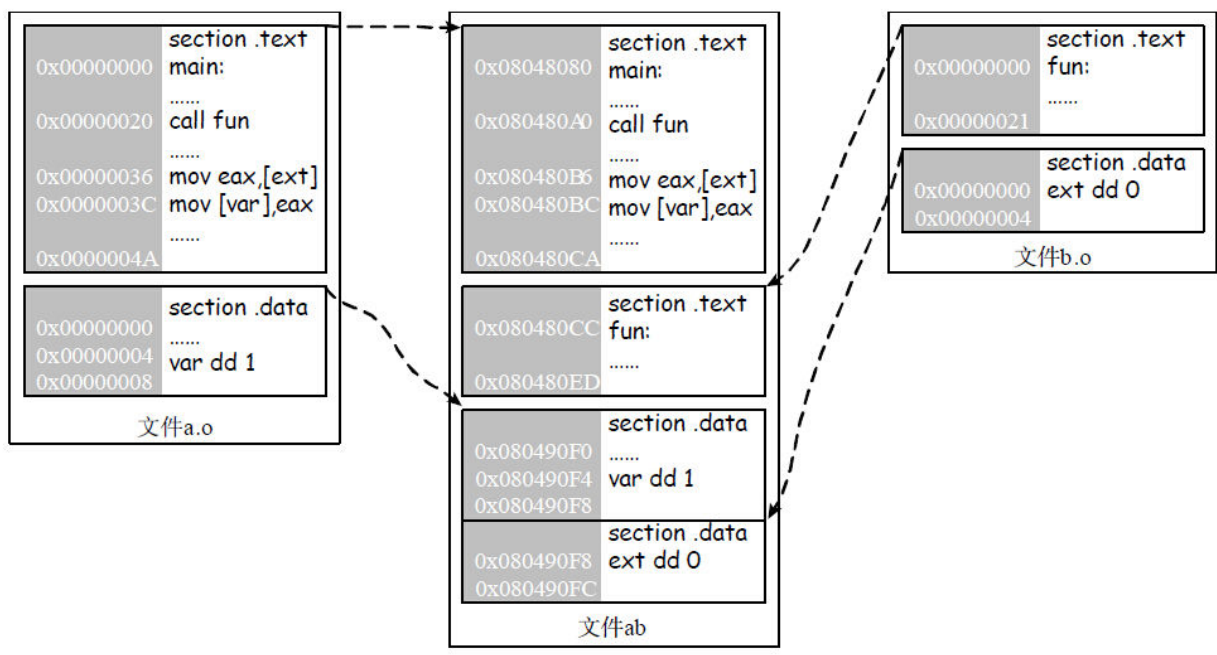


图7-3 地址空间分配

根据以上的描述，链接器对地址空间分配算法的实现为：

```
1 #define BASE_ADDR
   0x08048000                                     //默认加载地址

2 #define MEM_ALIGN
   4096                                           //默认内存对齐大小
```



```

3  #define DISC_ALIGN

4                                     //默认磁盘对齐大小

4  #define TEXT_ALIGN

16                                  //.text段对齐大小

5
6  void Linker::allocAddr

() {
7      unsigned int curAddr=BASE_ADDR;
8      unsigned int curOff=52+
9          sizeof(Elf32_Phdr)*segNames.size();
10     for(int i=0;i<segNames.size();++i) {
11         segLists[segNames[i]]->allocAddr(segNames[i],
12             curAddr,curOff);
13     }
14 }
15
16 void SegList::allocAddr

(string name,unsigned int& base,
17 unsigned int& off) {
18     begin=off;
19     //对齐前偏移

19
20     int align=DISC_ALIGN;
21     if(name==".text") align=TEXT_ALIGN;
22     off+=(align-off%align)%align;
23     base+=(MEM_ALIGN-base%MEM_ALIGN)%MEM_ALIGN
24         +off%MEM_ALIGN;
25
26     baseAddr=base;
27     //段基址

27     offset=off;
28     //对齐后偏移

28     size=0;
29     //段大小, 段内偏移

29
30     for(int i=0;i<ownerList.size();++i) {
31         Elf32_Shdr*seg=ownerList[i]->shdrTab[name];
32         int sh_align=seg->sh_align;
33         size+=(sh_align-size%sh_align)%sh_align;
34
35         char* buf=new char[seg->sh_size];           //段数据

36         ownerList[i]->getData(buf, seg->sh_offset, seg->sh_size);
37         blocks.push_back(new Block(buf, size, seg->sh_size));
38
39         seg->sh_addr=base+size;                       //段基址

40         size+=seg->sh_size;                           //累加段

```

内偏移

```
41 }
42 base+=size;
//累加基址

43 off+=size;
//累加偏移

44 }
45
46 void Elf_file::getData
(char*buf,Elf32_Off offset,
47 Elf32_Word size) {
48 FILE*fp=fopen(elf_dir,"rb");
49 rewind(fp);
50 fseek(fp,offset,0);
51 fread(buf,size,1,fp);
52 fclose(fp);
53 }
```

1) 第1~4行定义了链接器使用的常量：默认段加载起始地址、内存对齐大小、文件对齐大小、代码段对齐大小。

2) 第6~14行是为链接器进行地址空间分配的主流程。首先将 `curAddr` 初始化为段加载起始地址（0x08048000），`curOff` 为段起始文件偏移（ELF文件头+程序头表大小）。然后根据段名列表（包含“.text”和“.data”）对每个段类型进行地址空间分配。

3) 第16~44行处理每个类型段的地址空间分配。

4) 第18~28行首先将对齐前的文件偏移记录到 `begin` 字段，然后根据对齐字段 `align`（默认为4，处理代码段时设为16）修正文件偏移 `off`，接着使对齐段基址 `base` 与文件偏移字段相对于页大小取模的值相等，最

后将以上信息保存到SegList对象的字段中。其中size字段既表示合并后段的大小，也表示处理的目标文件的段偏移。

5) 第30~41行处理目标文件的段。首先取出被处理的段的段表项信息，根据段对齐字段sh_align对段偏移size对齐修正。然后调用getData函数从目标文件的sh_offset位置取出sh_size长度的数据到buf，基于这些信息构建Block对象并将其添加到数据块列表。计算段的基址并将其写回目标文件的段表项，将当前段的大小累加到段偏移size。

6) 第42~43行将合并后的段大小累加到基址字段base，同时将段大小累加到文件偏移字段off，为下一个类型的SegList的地址空间分配提供起始地址和偏移信息。

7) 第46~53行描述了getData函数的实现，即根据调用参数读取目标文件offset处开始长度为size的数据到缓存buf。

经过以上的处理，所有目标文件内需要加载的段基址都被计算出来，地址空间分配的工作结束。

7.3 符号解析

符号解析的主要目的是计算目标文件内符号的线性地址，即可执行文件被加载到进程内存空间之后符号的虚拟地址。目标文件符号表内保存了每个定义的符号相对于所在段基址的偏移，当段的地址空间分配结束后每个段的基址都被确定下来，因此符号地址可以使用如下公式计算：

符号地址=段基址+符号相对段基址的偏移

不过在计算符号地址之前，仍需要做一些准备工作。首先需要扫描目标文件内的符号表，获取符号的定义与引用的信息，即7.1节描述的导出符号集合和导入符号集合。其次，需要对导入符号集合和导出符号集合进行合法性验证。符号验证包含两个方面：

1) 符号重定义：即导出符号集合存在同名的符号。由于目标文件链接时，对符号的处理是按名检索的方式，符号重定义将导致引用该符号的文件无法确定应该具体使用哪个符号。

2) 符号未定义：即导入符号集合包含导出集合不存在的符号。当目标文件引用的外部符号在其他目标文件内找不到对应的定义时，就无法确定符号的地址。

一旦出现符号重定义或未定义的情况，链接器的工作就无法继续进行。因此，我们将符号解析分为两个阶段：符号引用验证和符号地址解析。只有符号引用验证通过后，才继续符号地址解析的流程。

图7-4描述了符号解析的流程。

1) 符号解析开始时，定义了两个空集。**Export**表示导出符号集合，**Import**表示导入符号集合。

2) 扫描所有目标文件符号表的所有符号，如果符号是导出符号且不在导出符号集合**Export**内，则将符号添加到**Export**集合，否则报告符号重定义错误，退出符号解析流程。如果符号是导入符号，则将符号添加到导入符号集合**Import**。

3) 所有的目标文件符号表扫描结束后，计算导入符号集合**Import**与导出符号集合的差集**Undef**。**Undef**集合记录了所有未定义的符号集合，只有该集合为空时才继续计算导入符号集合**Export**内符号的地址。否则**Undef**内的所有符号都是未定义符号，需要退出符号解析流程。

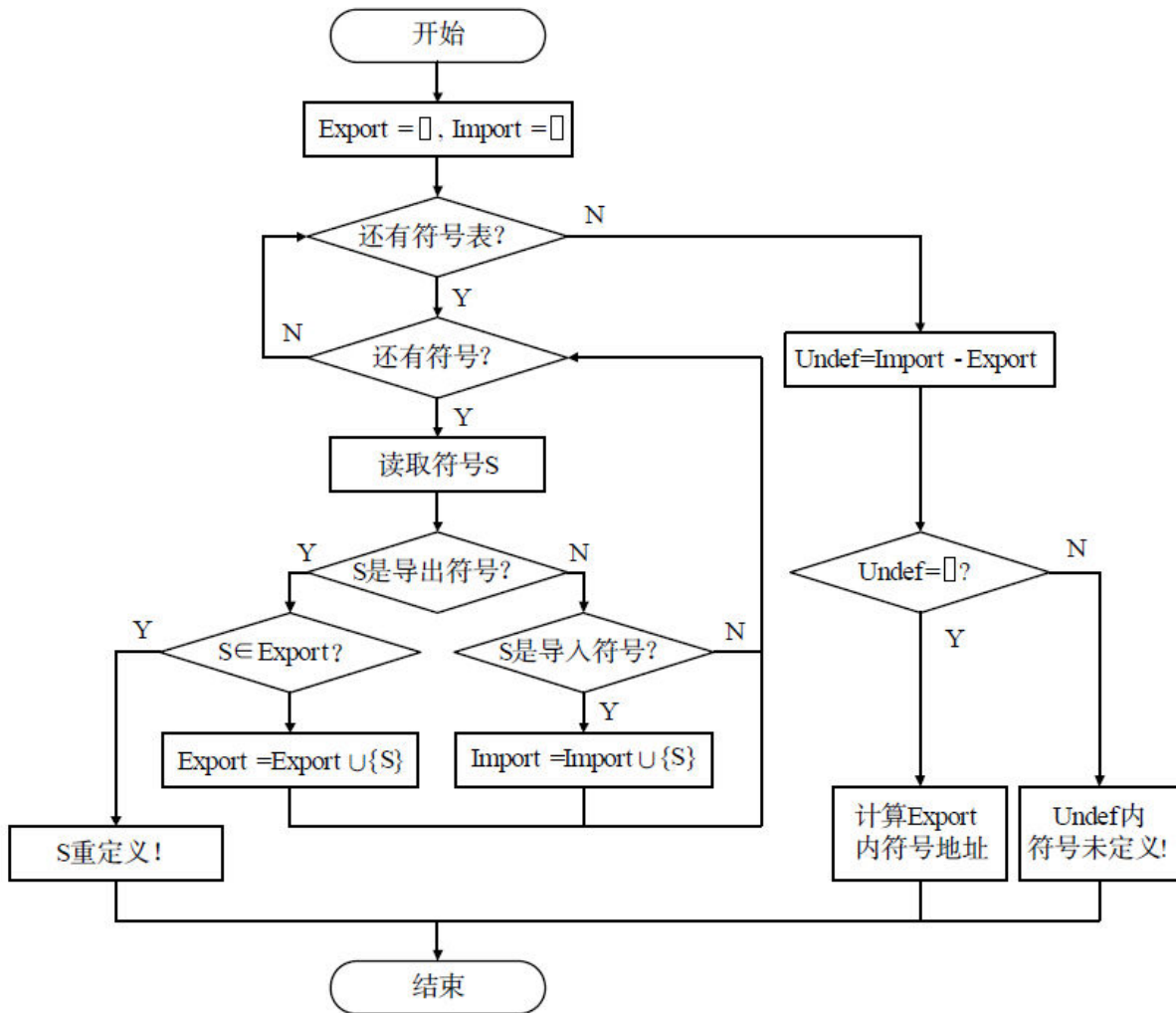


图7-4 符号解析流程

由于符号解析依赖于导入符号和导出符号这两个集合，首先看这两个集合的构造。

```

1 void Linker::collectInfo() {
2     for(int i=0;i<elfs.size();++i) {
3         Elf_file*elf=elfs[i];
4         //记录段表信息

5         for(int i=0;i<segNames.size();++i) {
6             if(elf->shdrTab.find(segNames[i])
7                != elf->shdrTab.end()){
8                 segLists[segNames[i]]->ownerList.push_back(elf);
9             }
10        }
  
```

```

11          //记录符号引用信息

12          for(hash_map<string,Elf32_Sym*,string_hash>::iterator
13              symIt=elf->symTab.begin();
14              symIt!=elf->symTab.end();++symIt) {
15              if(ELF32_ST_BIND(symIt->second->st_info)
16                  == STB_GLOBAL) {
17                  SymLink*symLink=new SymLink();
18                  symLink->name=symIt->first;          //符
号名

19              if(symIt->second->st_shndx==STN_UNDEF) {    //导
入符号

20                  symLink->recv=elf;

21                  symLink->prov=NULL;

22                  symLinks.push_back(symLink);
23              }
24              else {
//导出符号

25                  symLink->recv=NULL;

26                  symLink->prov=elf;

27                  symDef.push_back(symLink);
28              }
29          }
30      }
31  }
32 }

```

1) 函数collectInfo用于收集目标文件的信息。第3~10行扫描了目标文件的段信息，将ELF文件对象放入段地址空间分配使用的数据结构segList中。

2) 第11~30行扫描目标文件内符号的信息。对于每个ELF目标文件，只处理符号表中标识为STB_GLOBAL的全局类型的符号。对于符

号解析来说，并不关心局部符号的信息。这里岔开一个话题，在ELF目标文件的段表项数据结构中，符号表段表项的sh_info字段记录了符号表内第一个全局符号的索引。链接器可以直接从这个索引位置开始扫描符号表，获取所有的全局符号。不过我们在汇编器生成目标文件时，强制将该字段设为0，因此必须对符号表全部扫描。

3) 第18行记录符号名到symLink对象。

4) 第19~23行处理导入符号集合，即将引用该符号的ELF文件对象记录到symLink对象的recv字段，由于并不知道哪个目标文件定义了该符号，因此将prov字段设置为空，最后将symLink对象记录到导入符号集合symLinks。

5) 第24~28行处理导出符号集合，即将定义该符号的ELF目标文件对象记录到symLink对象的prov字段，由于一个符号可能会被多个目标文件引用，因此recv字段设置为空，最后将symLink对象记录到导出符号集合symDef。

有了导出符号集合和导入符号集合，接下来便可以进行符号引用验证的工作。

7.3.1 符号引用验证

根据前面对符号解析流程的描述，可以很容易实现符号引用验证算法。不过，在说明符号验证算法实现之前先说明一个细节问题。我们知道，目标文件和可执行文件有一个很大的区别：目标文件的文件头的程序入口点`e_entry`字段为0，而可执行文件的程序入口点是一个线性地址。这里我们需要先假定程序的入口地址被记录到一个名为“`@start`”的符号内，显然这个符号不可能是编译器生成的符号名。为了保证链接器可以找到程序入口点，那么符号引用验证阶段必须强制要求导出“`@start`”符号。至于“`@start`”符号的提供者，可以暂时认为来源于一个已有的目标文件。关于程序入口点的讨论会在7.5节详细展开。

基于以上的讨论，符号引用验证的算法实现为：

```
1  #define START
   "@start"
2
3  bool Linker::symValid
   () {
4      bool flag=true;
5      startOwner=NULL;
6      for(int i=0;i<symDef.size();++i) {
7          //记录入口点
8
9          if(symDef[i]->name==START
10             startOwner=symDef[i]->prov;
11             }
12         for(int j=i+1;j<symDef.size();++j) {
```

```

12                                     //符号重定义

13             if(symDef[i]->name==symDef[j]->name) {
14                 printf("symbol %s redefinition in %s and %s.\n",
15                     symDef[i]->name.c_str(),
16                     symDef[i]->prov->elf_dir,
17                     symDef[j]->prov->elf_dir
18                 );
19                 flag=false;
20             }
21         }
22     }
23     //找不到入口点

24     if(startOwner==NULL) {
25         printf("can not find entrypoint symbol %s.\n",START);
26         flag=false;
27     }
28     for(int i=0;i<symLinks.size();++i) {
29         for(int j=0;j<symDef.size();++j) {
30             //记录符号引用

31             if(symLinks[i]->name==symDef[j]->name) {
32                 symLinks[i]->prov=symDef[j]->prov;
33                 break;
34             }
35         }
36         //符号未定义

37         if(symLinks[i]->prov==NULL) {
38             printf("undefined symbol %s in %s.\n",
39                 symDef[i]->name.c_str(),
40                 symLinks[i]->recv->elf_dir
41             );
42             flag=false;
43         }
44     }
45     return flag;
46 }

```

1) 第7~10行在导出符号集合内搜索名为**START**的符号，如果找不到该符号则在第23~27行报告“找不到程序入口点”链接错误。

2) 第11~21行搜索导出集合内是否有重名的符号，一旦出现重名符号，则报告“符号重定义”链接错误，并指出符号冲突的目标文件。

3) 第28~44行处理符号引用的情况。第29~35行找出导出符号集合与导入符号集合都包含的符号，这属于符号引用找到了符号定义。因此将符号在导入符号集合的symLink对象的prov字段记录为定义符号的目标文件，即符号在导出符号集合的symLink对象的prov字段所指示的目标文件内。

4) 第36~43行处理符号未定义的情况。如果扫描完导出符号集合都没有找到被导入的符号，则报告“符号未定义”链接错误，并给出引用符号的目标文件。

符号引用验证函数symValid调用结束后，如果返回true则继续符号解析的流程，否则终止链接器的工作。

7.3.2 符号地址解析

符号引用验证过程中除了对导入符号集合和导出符号集合进行合法性验证之外，还“顺便”记录了定义导入符号的目标文件，以方便对符号地址进行解析。

具体来说，符号地址解析分为两个步骤：

- 1) 扫描所有ELF目标文件的本地符号，计算本地符号的地址。
- 2) 扫描所有导入集合的符号，将符号地址传递到引用该符号的目标文件的符号表内。

```
1 void Linker::symParser
2 {
3     for(int i=0;i<elfs.size();++i) {
4         Elf_file*elf=elfs[i];
5         for(hash_map<string,Elf32_Sym*,string_hash>::iterator
6             symIt=elf->symTab.begin();
7             symIt!=elf->symTab.end();++symIt) {
8             //所有本地符号
9
10            Elf32_Sym*sym=symIt->second;
11            if(sym->st_shndx!=STN_UNDEF) {
12                string segName=elf->shdrNames[sym->st_shndx];
13                sym->st_value+=elf->shdrTab[segName]->sh_addr;
14            }
15        }
16    }
17    for(int i=0;i<symLinks.size();++i) {
18        //所有符号引用
19
20        string name = symLinks[i]->name;
21        Elf32_Sym*provsym=symLinks[i]->prov->symTab[name];
22        Elf32_Sym*recvsym=symLinks[i]->recv->symTab[name];
23        recvsym->st_value=provsym->st_value;
```

```
22 }  
23 }
```

1) 函数`symParser`执行符号地址解析的流程。第2~14行计算每个目标文件的本地符号地址。即根据符号表项提供的段索引`st_shndx`找到对应的段表项，然后取出段基址`sh_addr`，累加到原有的符号地址（符号段偏移）`st_value`中，得到符号的线性地址。

2) 第16~22行处理符号引用的情况。符号引用验证函数`symValid`已经统计了每个符号引用所对应的符号引用文件和符号定义文件，并且在第一步中算出了所有已定义的符号的地址，因此直接取出目标文件中的符号表项`provsym`和`recvsym`，最后将`provsym`的符号地址传递给`recvsym`即可。

至此，得到了所有目标文件中符号的虚拟地址，符号解析工作完成。

7.4 重定位

回顾第6章对重定位表信息生成的描述，目标文件的重定位信息包含三个关键元素：重定位符号——使用哪个符号的地址进行重定位；重定位位置——在何处进行重定位；重定位类型——用何种方法进行重定位。

首先，由于重定位操作依赖于重定位符号的地址，因此在符号解析完成前是无法进行重定位的。重定位符号已经在收集ELF文件信息时记录到RelItem的relName字段内。

其次，重定位位置可以根据重定位段以及重定位位置的段内偏移计算得出。

重定位位置=重定位段基址+重定位位置的段内偏移

重定位段已经在收集ELF文件信息时记录到RelItem的segName字段内，根据段名可以取得段表项对象，继而获得段基址。至于重定位位置的段内偏移可以由RelItem的rel字段获得。

最后，重定位类型有两种：绝对地址重定位和相对地址重定位。根据不同的重定位类型对段数据进行修正操作是重定位的核心。

绝对地址重定位操作比较简单，需要绝对地址重定位的地方一般都是源于对符号地址的直接引用，由于汇编器不能确定符号的虚拟地址，最终使用0作为占位符填充了引用符号地址的地方。因此，绝对地址重定位操作只需要直接填写重定位符号的虚拟地址到重定位位置即可。

绝对重定位地址=重定位符号地址

相对地址重定位稍微复杂一点，需要相对地址重定位的地方一般都是源于跳转类指令引用了其他文件的符号地址。虽然汇编器不能确定被引用符号的虚拟地址，但是并不使用0作为占位符填充引用符号地址的地方，而是使用“重定位位置相对于下一条指令地址的偏移”填充该位置。链接器进行相对地址重定位操作时，会计算符号地址相对于重定位位置的偏移，然后将该偏移量累加到重定位位置保存的内容。这么说起来有点绕，其实本质上仅仅是计算的转换而已。

相对重定位地址=重定位符号地址-重定位位置+重定位位置数据内容

=（重定位符号地址-重定位位置）+（重定位位置-下一条指令地址）

=重定位符号地址-下一条指令地址

根据上面的计算，可以很清晰地看出最终计算的相对重定位地址正是符号地址相对于下一条指令地址的偏移，也正符合跳转类指令对操作数的要求。

至于为何对相对地址重定位进行如此“繁琐”的计算，笔者认为按照这样的方式，对于不同长度和设计结构的指令，只要重定位位置的数据按照相对地址的方式进行修正，那么相对重定位地址的计算方式不变，区别仅仅是重定位位置处的数据的值不同。比如对于Intel32位跳转指令该位置数据值是-4，对于Intel64位跳转指令该位置数据值是-8。

下面结合一个例子描述重定位的过程。

如图7-5所示，重定位表内有三个重定位项，第一项是相对地址重定位类型，剩余两项是绝对地址重定位类型。

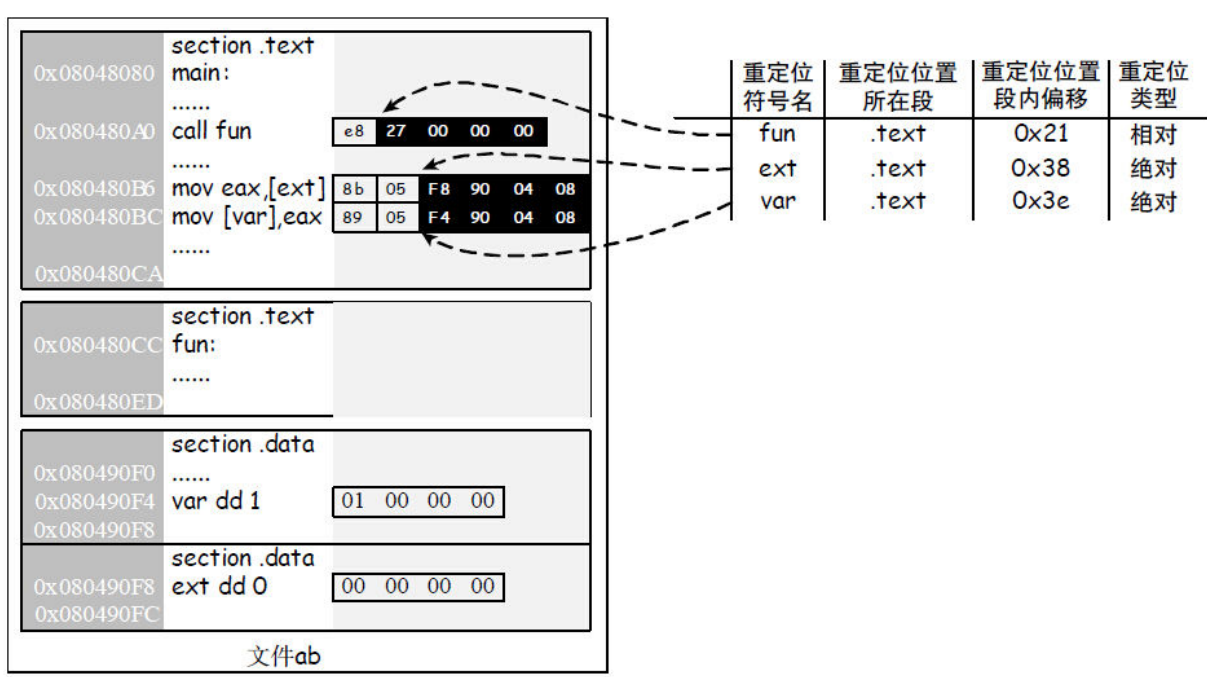


图7-5 重定位

对于第一个重定位项，重定位符号`fun`符号解析后的地址是`0x080480cc`，重定位位置是代码段“`.text`”的基址加上重定位位置的段内偏移，即`0x08048080+0x21=0x080480a1`，此处保存的数据值是`-4`。根据相对重定位地址的计算方法`0x080480cc-0x080480a1+(-4)=0x27`，即最终的`fun`地址相对于`call`指令下一条指令地址`0x080480a5`的偏移。

对于第二个重定位项，重定位符号`ext`符号解析后的地址是`0x080490f8`，重定位位置是代码段“`.text`”的基址加上重定位位置的段内偏移，即`0x08048080+0x38=0x080480b8`，此处保存的数据值是`0`。根据绝对重定位地址的计算方法，此处直接修改为`0x080490f8`，即`ext`的地址即可。类似地，对于第三个重定位项的处理不再赘述，需要注意的是，无论是绝对地址还是相对地址，都是以小字节序的方式保存的。

重定位操作的代码实现如下：

```
1 void Linker::relocate
2 ()
3 {
4     for(int i=0;i<elfs.size();++i) {
5         vector<RelItem*>tab=elfs[i]->relTab;
6         for(int j=0;j<tab.size();++j) {
7             //重定位符号
8
9             string relName=tab[j]->relName;
10            Elf32_Sym*sym=elfs[i]->symTab[relName];
11
12
13            string segName=tab[j]->segName;
14            Elf32_Shdr*seg=elfs[i]->shdrTab[segName];
```

```

14                                //重定位符号地址

15                                unsigned int symAddr
=sym->st_value;
16
17                                //重定位位置

18                                unsigned int offset=tab[j]->rel->r_offset;
19                                unsigned int relAddr

=sseg->sh_addr+offset;
20
21                                //重定位类型

22                                unsigned char type
=ELF32_R_TYPE(tab[j]->rel->r_info);
23
24                                segLists[segName]->relocAddr(relAddr, type, symAddr);
25                                }
26        }
27 }

```

- 1) 函数relocate遍历所有目标文件的重定位表relTab，取出每个重定位表项进行处理。
- 2) 第6~8行取出重定位符号名称relName，并根据符号表symTab得到符号表项sym。
- 3) 第10~12行取出重定位段名称segName，并根据段表shdrTab得到段表项seg。
- 4) 第14~15行从符号表项内读取符号的线性地址symAddr。
- 5) 第17~19行从重定位表项取出重定位位置的段内偏移，并根据重定位段的基址计算重定位位置的线性地址。

6) 第21~22行从重定位表项内取出重定位类型type。

7) 第24行调用SegList的relocAddr函数进行重定位操作。

```
1 void SegList::relocAddr
(
2     unsigned int relAddr,
3     unsigned char type,
4     unsigned int symAddr) {
5
6     // 查找修正地址所在位置

7     unsigned int relOffset=relAddr-baseAddr;
8     Block*block=NULL;
9     for(int i=0;i<blocks.size();++i)
10    {
11        unsigned int start=blocks[i]->offset;
12        unsigned int end=start+blocks[i]->size;
13        if(start <= relOffset && relOffset < end) {
14            block=blocks[i];
15            break;
16        }
17    }
18
19    // 地址修正

20    int *pAddr=(int*)(block->data+relOffset-block->offset);
21    if(type==R_386_32) { // 绝对地址修
22        *pAddr=symAddr;
23    }
24    else if(type==R_386_PC32) { // 相对地址修
25        *pAddr=symAddr-relAddr+*pAddr;
26    }
27 }
```

1) 函数relocAddr执行重定位操作，即根据重定位项描述的信息修正段内二进制数据。

2) 第6~17行查询重定位位置的地址所在的数据块Block。

3) 第7行计算出重定位位置相对于合并后段基址的偏移relOffset。

4) 第11~12行计算每个Block数据块的起始地址start和结束地址end，通过比较relOffset是否落在[start, end) 地址区间内以查询重定位位置所在的数据块。查询得到的数据块记录到block变量中。由于重定位表项由汇编器生成，因此必然存在一个数据块满足查询条件，即block不会为NULL。

5) 第19~26行进行重定位操作。由于block的offset字段记录了block相对于合并后段基址的偏移，relOffset记录了重定位位置相对于合并后段基址的偏移，因此两者之差便是重定位位置相对于重定位数据块起始地址的偏移。根据block保存的数据块的缓冲区地址data和该偏移可以得到重定位位置在缓冲区内的地址，将该地址记录到int指针pAddr。

6) 第21~23行进行绝对地址修正。即将符号地址symAddr写入pAddr指向的内存区域。通过这样的方式“恰好”将pAddr指向的数据按照小字节序修改为symAddr的值。

7) 第24~26行进行相对地址修正。即将符号地址symAddr减去重定位位置的地址relAddr，然后累加pAddr指向的数据区域原本保存的值（-4），得到最终的相对地址，并将其写回到pAddr指向的数据区域。

重定位操作结束后，链接器的核心工作已全部完成。

7.5 程序入口点与运行时库

在7.3.1节曾提到程序入口点地址被保存在一个名为“@start”的特殊符号内，而定义该符号的目标文件并不是编译器根据源代码生成的。那么这就有两个问题需要弄清楚：

- 1) 为什么引入新的符号而不是main函数作为程序入口点？
- 2) 定义新的符号的目标文件该如何得到？

首先解释第一个问题。根据第3章代码生成的逻辑，对于main函数生成的汇编代码片段形式如下：

```
1  global main
2  main:

3      push ebp
4      mov  ebp, esp
5      sub  esp, ?
6      ...
7      mov  esp, ebp
8      pop  ebp
9      ret
```

本质上讲，main函数与普通的函数并没有太大的区别：包含函数入栈代码（第3~5行）、函数体代码（第6行省略内容）和函数出栈代码（第7~9行）。假定使用main函数作为程序入口点，即将main符号的线性地址写入ELF文件头部的e_entry字段，那么程序加载运行后会从

`main`符号的地址位置读取指令开始执行。整个`main`函数执行过程不会出现任何问题，直到`ret`指令执行结束后。根据`ret`指令的语义，程序会从栈顶取出32位的数据作为返回地址，然后跳转到该地址继续执行！然而，程序执行`main`函数之前，栈顶保存的数据是未知的，因此导致程序的最终行为无法预测，最常见后果是触发进程“Segment Fault”。

因此，为了让程序可以“优雅”地退出，必须构造一个`main`函数的调用者完成函数调用后的“清理”工作。这也为第二个问题提供了解决办法。

```
1  global @start
2  @start:

3      ...
4      call main
5      ...
6      mov eax, 1
7      mov ebx, 0
8      int 128
```

在Linux的系统调用中，调用号为1的系统调用是`exit`，使用`exit`可以使进程正常退出。调用`exit`的汇编代码如第6~8行所示，其中寄存器`eax`保存`exit`系统调用号1，`ebx`保存`exit`系统调用的参数0，`int`指令触发`exit`系统调用退出进程。

符号“`@start`”处的代码会调用`main`函数后使用`exit`系统调用退出进程，在调用`main`函数前后可以执行一些初始化工作（第3行省略内容）

和清理工作（第5行省略的内容）。由于我们设计的编译器main函数的定义很简单，因此并不需要初始化和清理的操作。

将上述代码保存在start.s，并使用我们实现的汇编器处理后，可以得到目标文件start.o。然后，使用readelf工具查看start.o的符号表。

Symbol table '.symtab' contains 3 entries:							
Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000		0 NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000		0 NOTYPE	GLOBAL	DEFAULT	1	
@start							
2:	00000000		0 NOTYPE	GLOBAL	DEFAULT	UND	
main							

可以发现目标文件start.o包含一个导出符号“@start”，一个导入符号“main”。将start.o和汇编器生成的目标文件一起交给链接器处理，如果其他目标文件没有定义main函数，则会触发“main符号未定义”链接错误。至于链接器生成的可执行文件的程序入口点，自然是设置为符号“@start”的线性地址即可。

如图7-6所示，从整个编译系统的工作流程来看，start.o文件是编译系统正常工作必需的目标文件。无论编译系统处理的源代码如何定义，在最终的链接阶段必须将start.o和其他目标文件一起链接才能正常生成可执行文件。对于这样的目标文件，有一个统一的名称——“语言运行时库”。显然，start.o应该是最简单的运行时库了，它只负责引导调用main函数，别的什么也没做。

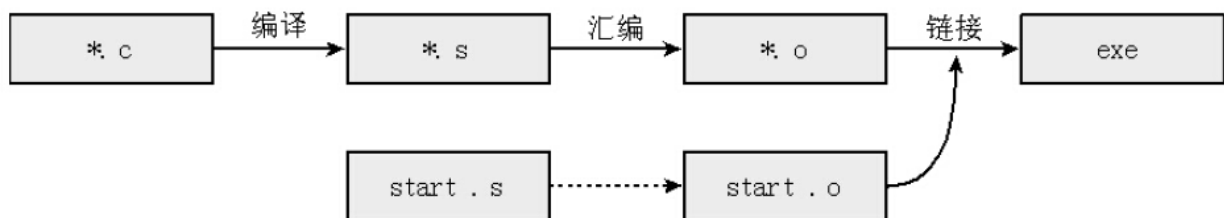


图7-6 编译系统工作流程

根据类似的方式，可以很方便地扩展程序设计语言运行时库的功能。比如可以定义`printf.s`实现标准输出函数`printf`，经过汇编器处理后生成`printf.o`目标文件。只需要源码声明使用了`printf`函数，在链接时将`printf.o`链接到可执行文件，即可在高级语言中实现标准输出的功能。更进一步地，可以直接定义`math.c`文件实现数学相关的函数，经过编译器和汇编器处理后生成`math.o`目标文件，这样高级语言就可以进行复杂的数学计算。

如果在编译系统中实现了预处理器并支持`include`指令，那么像`printf`函数或`math.c`实现的函数声明语句就可以放入类似“`stdio.h`”或“`math.h`”这样的头文件内。如果链接器支持输入压缩包格式的文件，那么像`printf.o`和`math.o`这样的目标文件可以打包放在类似“`libc.a`”这样的压缩包中，链接器只需要在链接之前将压缩包解压即可。编写高级语言程序时，只要包含需要的头文件，并在链接阶段包含对应的库文件，就可以使用更强大的语言特性。不知不觉，我们发现这样的实现方式已经与GCC非常接近了。

相比而言，GCC的C语言运行时库（C Runtime Library, CRT）复杂得多。回顾第1章描述GCC静态链接工作流程时涉及的5个目标文件 `crt1.o`、`crti.o`、`crtbeginT.o`、`crtend.o`、`crtfn.o`，以及3个静态库 `libgcc.a`、`libgcc_eh.a`、`libc.a`，这些文件的功能分别为：

1) `crt1.o`：定义程序入口点“`_start`”、调用“`.init`”段的代码执行程序
的初始化、调用`main`函数、调用“`.fini`”段的代码执行程序
的清理操作。
早期版本为`crt0.o`，不支持“`.init`”和“`.fini`”段。

2) `crti.o`：定义“`.init`”段的函数入栈代码、调用C++全局构造代码。

3) `crtfn.o`：定义“`.fini`”段的函数出栈代码、调用C++全局析构代码。

4) `crtbeginT.o`：定义C++全局构造代码。

5) `crtend.o`：定义C++全局析构代码。

6) `libc.a`：定义C语言标准库代码。

7) `libgcc.a`：定义由于平台差异性的辅助函数代码。

8) `libgcc_eh.a`：定义C++异常处理的平台相关代码。

由此可见，对于一种高级语言，除了编译器、汇编器和链接器是
必不可少的部分之外，语言的运行时库也是不可或缺的一部分。虽然

我们实现的编译系统旨在弄清高级语言实现的细节，但是绝不能忽略语言运行时库的地位。功能丰富的运行时库，可以让高级语言的表达能力更加强大。

7.6 可执行文件生成

链接器的最终输出是ELF格式的可执行文件。第5章描述了ELF文件的通用结构，而在链接器生成可执行文件时，只关心可执行文件的ELF文件结构。

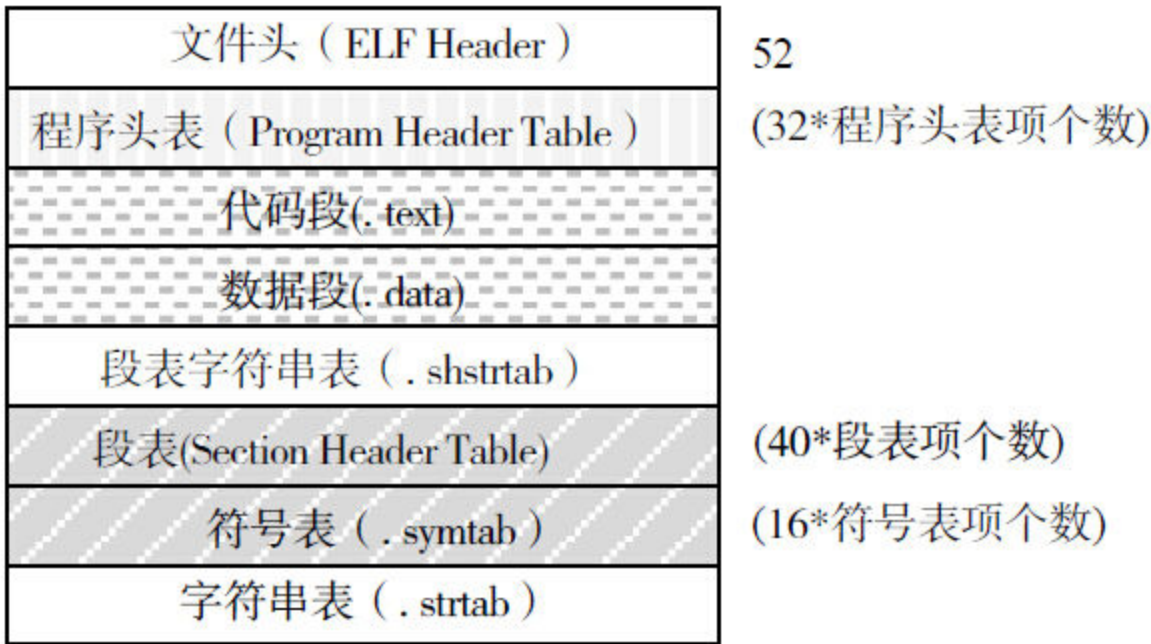


图7-7 ELF可执行文件

如图7-7所示，在静态链接器生成的可执行文件中，不会包含重定位表。程序头表是ELF可执行文件独有的结构，记录需要加载到进程地址空间的段。代码段来源于所有的目标文件的代码段拼接重定位后的数据块，数据段来源于所有的目标文件的数据段拼接重定位后的数

据块。符号表段来源于所有目标文件定义的全局符号，其中符号表项的`st_shndx`字段需要根据最终生成的段表结构更新。另外，ELF文件头、段表、段表字符串表、字符串表在ELF文件结构确定后可以计算得出。

与汇编器的目标文件生成的流程相似，可执行文件的生成也可以分为两个阶段。

1) ELF文件结构组装。根据链接器的处理得到ELF可执行文件的信息，生成ELF文件结构。包括文件头各个字段的值、串表的内容以及引用串表内的字符串偏移信息（如段表项的段名字段`sh_name`、符号表项的符号名字段`st_name`）、符号表项的`st_shndx`字段等。

2) ELF文件结构输出。输出文件头、程序头表、代码段、数据段、段表字符串表、段表、符号表、字符串表的内容。任何两个文件结构若因为对齐需要产生了空隙，则使用0填充补齐。

首先看ELF可执行文件结构的组装。

```
1 void Elf_file::assemObj
  (Linker* linker) {
2     //所有段名

3     vector<string> AllSegNames;
4     AllSegNames.push_back("");
5     vector<string> segNames = linker->segNames;
6     for (int i=0;i<segNames.size();++i){
7         AllSegNames.push_back(segNames[i]);
8     }
9     AllSegNames.push_back(".shstrtab");
10    AllSegNames.push_back(".symtab");
```

```

11     AllSegNames.push_back(".strtab");
12
13     // 段索引

14     hash_map<string,int,string_hash> shIndex;
15     // 段名索引

16     hash_map<string,int,string_hash> shstrIndex;
17     // 建立索引

18     for (int i=0;i<AllSegNames.size();++i){
19         string name = AllSegNames[i];
20         shIndex[name] = i;
21         shstrIndex[name] = shstrtab.size();
22         shstrtab += name;
23     }
24     // 保存数据

25     shstrtab.push_back('\0');
26     // 生成符号表

27     addSym("",NULL);
28     vector<SymLink*>symDef = linker->symDef;
29     for (int i=0;i<symDef.size();++i){
30         string name = symDef[i]->name;
31         Elf_file* prov = symDef[i]->prov;
32         Elf32_Sym*sym = prov->symTab[name];
33         String segName = prov->shdrNames[sym->st_shndx];
34         sym->st_shndx = shIndex[segName];
35         addSym(name,sym);
36     }
37     // 符号索引

38     hash_map<string,int,string_hash> symIndex;
39     // 符号名索引

40     hash_map<string,int,string_hash> strIndex;
41     // 建立索引

42     for (int i=0;i<symNames.size();++i){
43         string name = symNames[i];
44         symIndex[name] = i;
45         strIndex[name] = strtab.size();
46         strtab += name;
47     }
48     // 保存数据

49     strtab.push_back('\0');
50     // 更新符号表符号名索引

51     for (int i=0;i<symNames.size();++i){
52         string name = symNames[i];
53         symTab[name]->st_name=strIndex[name];
54     }

```

```

55     }
56
57     //处理文件头

58     char magic[] = {
//魔数

59         0x71, 0x45, 0x4c, 0x46,
60         0x01, 0x01, 0x01, 0x00,
61         0x00, 0x00, 0x00, 0x00,
62         0x00, 0x00, 0x00, 0x00
63     };
64
65     memcpy(&ehdr.e_ident, magic, sizeof(magic));
66     ehdr.e_type=ET_EXEC;
67     ehdr.e_machine=EM_386;
68     ehdr.e_version=EV_CURRENT;
69     ehdr.e_entry=symTab[START]->st_name;
70     ehdr.e_phoff=0;
71     ehdr.e_shoff=0;
72     ehdr.e_flags=0;
73     ehdr.e_ehsize=sizeof(Elf32_Ehdr);
74     ehdr.e_phentsize=sizeof(Elf32_Phdr);
75     ehdr.e_phnum=segNames.size();
76     ehdr.e_shentsize=sizeof(Elf32_Shdr);
77     ehdr.e_shnum=AllSegNames.size();
78     ehdr.e_shstrndx=shIndex[".shstrtab"];
79
80     int curOff = sizeof(ehdr); //文
//件头, 已对齐

81
82     ehdr.e_phoff = curOff;
//程序头表偏移

83
84     //生成程序头表, 已对齐

85     for(int i=0;i<segNames.size();++i){
86         string name=segNames[i];
87         Elf32_Word flags=PF_W|PF_R;
88         if(name==".text")flags=PF_X|PF_R;
89         addPhdr(PT_LOAD, segLists[name]->offset,
90                 segLists[name]->baseAddr,
91                 segLists[name]->size, segLists[name]->size,
92                 flags, MEM_ALIGN);
93     }
94     curOff += e_phentsize*e_phnum;
95
96     //生成已有段表

97     addShdr("", 0, 0, 0, 0, 0, 0, 0, 0); //空段表
//项

98
99     for(int i=0;i<segNames.size();++i){
100         string name=segNames[i];
101         Elf32_Word sh_flags=SHF_ALLOC|SHF_WRITE;
102         Elf32_Word sh_align=DISC_ALIGN;
103         if(name==".text"){

```

```

103             sh_flags=SHF_ALLOC|SHF_EXECINSTR;
104             sh_align=TEXT_ALIGN;
105         }
106         addShdr(name, SHT_PROGBITS, sh_flags,
107             segLists[name]->baseAddr,
108             segLists[name]->offset,
109             segLists[name]->size,
110             0, 0, sh_align, 0);
111         curOff=segLists[name]->offset +
112             segLists[name]->size;
113     }
114     curOff += (4-curOff%4)%4;                                     //
    对齐

115
116     //添加新的段表项

117     addShdr(".shstrtab", SHT_STRTAB, 0, 0, curOff,
118         shstrtab.size(), SHN_UNDEF, 0, 1, 0);
119     curOff += shstrtab.size();
120     curOff += (4-curOff%4)%4;                                     //
    对齐

121
122     ehdr.e_shoff = curOff;                                         //段表
    偏移

123     curOff += ehdr.e_shnum*ehdr.e_shentsize;                     //段表, 已对齐

124
125     addShdr(".symtab", SHT_SYMTAB, 0, 0, curOff,
126         symNames.size()*sizeof(Elf32_Sym),
127         shIndex[".strtab"], 0, 1, sizeof(Elf32_Sym));
128     curOff += symNames.size()*sizeof(Elf32_Sym);                 //已对齐

129
130     addShdr(".strtab", SHT_STRTAB, 0, 0, curOff,
131         strtab.size(), SHN_UNDEF, 0, 1, 0);
132     curOff += strtab.size();
133     curOff += (4-curOff%4)%4;                                     //
    对齐

134
135     //更新段表段名索引

136     for (int i=0; i<AllSegNames.size(); ++i){
137         string name = AllSegNames[i];
138         shdrTab[name]->sh_name=shstrIndex[name];
139     }
140 }

```

1) 第2~11行, 我们使用**AllSegNames**记录所有的段名, 包括空段、**segNames**记录的段 (需要加载的代码段“.text”和数据段“.data”), 以及ELF文件内部使用的段: “.shstrtab”“.symtab”“.strtab”。

2) 第13~24行建立段索引**shIndex**和段名索引**shstrIndex**。 **shIndex**记录了每个段表项在**AllSegNames**的索引位置, 段表会根据**AllSegNames**记录的段名顺序生成各个段表项。 **shstrIndex**记录了每个段名在段表字符串表“.shstrtab”内的位置, 我们按**AllSegNames**的顺序拼接所有的段名形成段表字符串表, 拼接时注意使用‘\0’分割不同的段名。

3) 第26~36行根据所有导出符号集合**symDef**生成符号表 (首先添加空符号表项)。通过每个**SymLink**的**prov**字段得到定义导出符号的ELF文件, 取出对应的符号表项。并根据符号表项内的**st_shndx**字段取出符号在目标文件内所在的段名。最后根据获取该段名对应的段表项在可执行文件段表内的索引, 更新符号对象的**st_shndx**字段。

4) 第38~49行建立符号索引**symIndex**和符号名索引**strIndex**。 **symIndex**记录了每个符号表项 (包括初始化时添加的空符号表项) 在符号名列表**symNames**的索引位置, 符号表会根据**symNames**记录的符号名顺序生成各个符号表项。 **strIndex**记录了每个符号名在字符串

表“.strtab”内的位置，我们按symNames的顺序拼接所有的符号名形成字符串表，拼接时注意使用‘\0’分割不同的符号名。

5) 第51~55行扫描符号表strTab，然后将每个符号表项的st_name字段设为符号名在字符串表内的索引，完成符号表信息的补充。

6) 第57~78行处理ELF文件头，按照第5章描述的ELF文件头的内容设置对应字段。其中，e_type设置为ET_EXEC表示文件类型是可执行文件，e_entry设置为符号START的线性地址表示程序入口地址，e_ehsize设置为sizeof (Elf32_Ehdr) 表示文件头大小，e_shentsize设置为sizeof (Elf32_Shdr) 表示段表项的大小，e_phentsize设置为sizeof (Elf32_Phdr) 表示程序头表项的大小，e_shnum设置为AllSegNames的大小表示段表项个数，e_phnum设置为segNames的大小表示程序头表项的个数，e_shstrndx设置为“.shstrtab”的段索引。另外，e_shoff和e_phoff暂时初始化为0，因为还未确定段表和程序头表的文件偏移。

7) 第80~82行将curOff初始化为ELF文件头的大小，并将e_phoff设为curOff，即程序头表起始位置。

8) 第84~94行生成程序头表。通过遍历segNames获取需要加载的段名，并从segLists取出合并的段列表。对于代码段“.text”，程序头表项的p_flags字段设为PF_X|PF_R，即可读可执行。对于其他段默认设置为PF_W|PF_R，即可读可写。程序头表项的p_offset、p_vaddr、和

p_memsz字段可以从SegList对应字段取出。另外p_type字段设为PT_LOAD表示段需要加载，p_align字段设为MEM_ALIGN表示被加载段在内存中以4KB对齐。最后，将curOff与程序头表的大小累加（程序头表大小=程序头表项个数×程序头表项大小）。

9) 第96~114行生成空段表项和可加载的段表项。通过遍历segNames获取需要加载的段名，并从segLists取出合并的段列表。对于代码段“.text”，段表项的sh_flags字段设为SHF_ALLOC|SHF_EXECINSTR，表示可分配可执行，sh_align字段设为TEXT_ALIGN，表示段按照16字节对齐。对于其他段，段表项的sh_flags字段设为SHF_ALLOC|SHF_WRITE，表示可分配可读写，sh_align字段设为DISC_ALIGN，表示段按照4字节对齐。段表项的sh_offset、sh_addr、和sh_size字段可以从SegList对应字段取出。另外sh_type字段设置为SHT_PROGBITS，表示段内数据是程序数据。最后，将curOff按4字节对齐。

10) 第116~120行添加“.shstrtab”的段表项到段表shdrTab。其中比较关键的字段有段名“.shstrtab”、段类型SHT_STRTAB、段文件偏移curOff、段大小shstrtab.size（）、对齐大小1（即该段的文件偏移不进行对齐）等。然后将当前段大小与curOff累加、对齐（后续文件结构要求的对齐方式），继续处理下一个文件结构。

11) 第122~123行处理段表的信息。首先将文件头的e_shoff设为curOff，即段表的偏移。然后将段表的大小与curOff累加（段表的大小=段表项个数×段表项大小）。

12) 第125~128行添加“.symtab”的段表项到段表shdrTab。其中比较关键的字段有段名“.symtab”、段类型SHT_SYMTAB、段文件偏移curOff、段大小（符号表项个数×符号表项大小）、sh_link记录符号表使用的串表“.strtab”的段索引、sh_info记录第一个全局符号的符号索引（由于我们没有在符号表内区分全局符号的区域，因此设为0，这样链接器会扫描所有的符号，根据符号的全局属性进行符号解析）、对齐大小1（即该段的文件偏移不进行对齐）、符号表项大小sizeof（Elf32_Sym）等。然后将当前段大小与curOff累加，继续处理下一个文件结构。

13) 第130~133行添加“.strtab”的段表项到段表shdrTab。其中比较关键的字段有段名“.strtab”、段类型SHT_STRTAB、段文件偏移curOff、段大小strtab.size（）、对齐大小1（即该段的文件偏移不进行对齐）等。然后将当前段大小与curOff累加、对齐（后续文件结构要求的对齐方式），继续处理下一个文件结构。

14) 第135~139行扫描段表shdrTab，然后将每个段表项的sh_name字段设为段名在段表字符串表内的索引，完成段表信息的补充。

到这里，已经完成了ELF可执行文件信息的组装。

最后，便是ELF可执行文件结构的输出。

```
1 void Elf_file::writeElf
  (Linker* linker){
2     int padNum = 0;
3     char pad[1]={0};
4
5         //文件头
6
7         fwrite(&ehdr,ehdr.e_ehsize,1,fout);
8         //程序头表
9
10        for(int i=0;i<phdrTab.size();++i){
11            fwrite(phdrTab[i],ehdr.e_phentsize,1,fout);
12        }
13 // .text .data
14 for(int i=0;i<segNames.size();++i) {
15     SegList*segs=linker->segLists[segNames[i]];
16     padNum=segs->offset - segs->begin;
17     fwrite(pad,sizeof(pad),padNum,fout);
18
19     Block*last=NULL;
20     for(int j=0;j<segs->blocks.size();++j){
21         Block*block=segs->blocks[j];
22         if(last!=NULL){
23             lastEnd = last->offset + last->size;
24             padNum = block->offset - lastEnd;
25             fwrite(pad,sizeof(pad),padNum,fout);
26         }
27         fwrite(block->data,block->size,1,fout);
28     }
29 }
30
31 // .shstrtab
32 padNum = shdrTab[".shstrtab"]->sh_offset
33         - shdrTab[".data"]->sh_offset
34         - shdrTab[".data"]->sh_size;
35 fwrite(pad,sizeof(pad),padNum,fout);
36 fwrite(shstrtab.c_str(),shstrtab.size(),1,fout);
37
38 //段表
39 padNum = ehdr.e_shoff
40         - shdrTab[".shstrtab"]->sh_offset
41         - shdrTab[".shstrtab"]->sh_size;
42 fwrite(pad,sizeof(pad),padNum,fout);
43 for(int i=0;i<shdrNames.size();++i){
44     Elf32_Shdr*sh=shdrTab[shdrNames[i]];
45     fwrite(sh,ehdr.e_shentsize,1,fout);
46 }
47 }
```

```
48 //符号表
```

```
49 for(int i=0;i<symNames.size();++i){  
50     Elf32_Sym*sym=symTab[symNames[i]];  
51     fwrite(sym,sizeof(Elf32_Sym),1,fout);  
52 }  
53  
54 //.strtab  
55 fwrite(strtab.c_str(),strtab.size(),1,fout);  
56 }
```

1) 首先输出ELF文件头ehdr，调用fwrite将该结构输出到文件即可。

2) 第8~11行输出程序头表。

3) 第13~29行输出可加载段，即代码段和数据段的二进制内容。通过遍历所有的可加载段名，从segLists中获取合并后段列表segs。对每一个segs，计算offset字段和begin字段的差值，使用0填充段列表对齐产生的缝隙。对每个段列表segs，通过遍历段列表内的每个数据块block，计算当前数据块起始位置与上一个数据块last结束位置的差值，即数据块因为对齐产生的缝隙，并使用0填充，最后输出每个数据块的内容。

4) 第31~36行输出“.shstrtab”段。首先使用0填充“.shstrtab”段和“.data”段因为对齐产生的间隙。然后输出shstrtab保存的段名字符串内容。

5) 第38~46行输出段表。首先使用0填充段表和“.shstrtab”段因为对齐产生的间隙。然后按照shdrNames保存的段名顺序输出shdrTab保

存的段表项内容。

6) 第48~52行输出“.symtab”段。只需按照symNames保存的符号名顺序输出symTab保存的符号表项内容即可。

7) 第54~56行输出“.strtab”段。只需输出strtab保存的符号名字符串内容即可。

至此，我们将可执行文件的内容输出完毕，完成了链接器的最后一步操作。使用readelf命令，可以查看验证我们输出的可执行结构的正确性。

最后，激动人心的时刻终于到来了！使用chmod+x命令为链接器输出的ELF文件添加可执行文件权限，并在shell内执行，就可以看到我们用自定义语言编写的代码的执行效果了。

7.7 本章小结

本章我们根据已设计的链接器结构，分别从信息收集、地址空间分配、符号解析、重定位、可执行文件生成的角度描述了一个简单的静态链接器的实现。另外，在处理程序入口点的问题时，我们还讨论了高级语言运行时库对语言的重要意义。

从实现角度来看，链接器并不像编译器和汇编器那样的语言翻译程序，而更像ELF文件的处理程序。如果说编译器做的事情是文本到文本的转换，那么汇编器则是文本到二进制的转换，而对于链接器则是二进制到二进制的转换。从编译器开始起步，到汇编器的中间过渡，最后到链接器的合并收尾，整个编译系统的工作流程是一个不可分割的整体。每一个流程都会对后续的操作产生影响，每一个结果生成的细节也会反作用于最初的设计。

回顾整个编译系统实现的细节，我们发现除了横向讨论的编译系统实现的每一个流程之外，还有很多纵向的信息值得深思。比如处理文法的自动机和文法理论、各种各样的数据结构和算法、指令系统的设计、可执行文件结构等，都是计算机学科的话题。笔者希望通过描述一个相对完整的编译系统实现，帮助读者更好地理解计算机软件的工作原理，基于此反思计算机学科每个知识领域的现实意义。

参考文献

[1]Alfred V Aho, Monica S Lam, Ravi Sethi, Jeffrey D Ullman.编译原理——原理、技术与工具[M].赵建华, 郑滔, 戴新宇, 译.北京: 机械工业出版社, 2008.

[2]俞甲子, 石凡, 潘爱民.程序员的自我修养——链接、装载与库[M].北京: 电子工业出版社, 2009.

[3]Keith Cooper, Linda Torczon.编译器设计[M].郭旭, 译.北京: 人民邮电出版社, 2012.

[4]John R Levine.链接器和加载器[M].李勇, 译.北京: 北京航空航天大学出版社, 2009.

[5]Andrew W Appel.现代编译原理: C语言描述[M].赵克佳, 黄春, 沈志宇, 译.北京: 人民邮电出版社, 2006.

[6]Kip Irvine.Intel汇编语言程序设计[M].温玉杰, 梅广宇, 罗云彬, 等译.5版.北京: 电子工业出版社, 2007.

[7]Intel® 64 and IA-32 Architectures Software Developer's Manual.

[8]Christopher W Fraser, David R Hanson.可变目标C编译器：设计与实现[M].王挺，译.北京：电子工业出版社，2005.

[9]裘巍.编译器设计之路[M].北京：机械工业出版社，2011.

[10]Steven S Muchnick.高级编译器设计与实现[M].赵克佳，沈志宇，译.北京：机械工业出版社，2005.

[11]张素琴.编译原理[M].北京：清华大学出版社，2011.